DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/DS/ENG/99-02

TIMED SAFETY AUTOMATA

AND

LOGIC CONFORMANCE

DISSERTATION
Frank Charles Duane Young
Major, USAF

AFIT/DS/ENG/99-02

AFIT/DS/ENG/99-02

TIMED SAFETY AUTOMATA AND LOGIC CONFORMANCE

DISSERTATION

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

Frank Charles Duane Young, B.S., M.S.C.E.
Major, USAF

June, 1999

Approved for public release; distribution unlimited

AFIT/DS/ENG/99-02

TIMED SAFETY AUTOMATA
AND
LOGIC CONFORMANCE

Frank Charles Duane Young, B.S., M.S.C.E.

Major, USAF

Approved:

Thomas C. Hartrum — 1 June 99

Thomas C. Hartrum, Chairman

Ken — 1 Jun 99

Kenneth S. Stevens

Robert P. Graham — 1 Jun 99

Robert P. Graham Jr.

Mark E. Oxley — 1 June 99

Mark E. Oxley

Larry W. Burggraf — 1 June 99

Larry W. Burggraf, Dean's Representative

Robert A. Calico, Jr

Dean, Graduate School of Engineering

Acknowledgements

I dedicate this dissertation to my wife Sharon and our children Lonnie, Andrew, Jonathan, Rachel, and Michael. Without their loving support and encouragement, I would have given up long ago. I deeply appreciate the wise counsel and guidance of the three committee chairmen, Lt. Col Paul Bailor, Dr. Ken Stevens, and Dr. Thomas Hartrum who have patiently guided me through the rough waters of my Ph.D. research at different times during the last six years. I sincerely appreciate committee members Dr. Mark Oxley and Major Robert Graham Jr. for their support over the long haul. I especially thank Dr. Graham for his rigorous and patient reflection during our many proof-sessions; without him, I would never have been able to see the forest for the sake of the trees. Thanks also to Mr. Jimmy Williamson and Mr. Darrell Barker, my Air Force Research Laboratory supervisors who steadfastly supported continuing my research. Finally, thanks be to God, with whom all things are possible—even AFIT doctorates.

Frank Charles Duane Young

Table of Contents

Page

List of Figures

Figure Page

List of Tables

List of Definitions

Definition Page

Definition Page

AFIT/DS/ENG/99-02

Abstract

Timed Logic Conformance (TLC) is used to verify the behavioral and timing properties of detailed digital circuits against abstract circuit specifications when both are modeled as Timed Safety Automata (TSA) with real-valued clocks. TLC is a bisimulation-style partial order relationship defined over TSA state space. In contrast to timed simulation, Calculus of Timed Refinement, and time-abstracted bisimulation, TLC defines when one system is an acceptable implementation of another by asymmetric action-matching requirements for specification inputs and implementation outputs. TLC intuitively and pragmatically supports writing abstract specifications and verifying them against implementations. TLC scales up by substituting verified specifications for implementations and hierarchically verifying larger systems. The TLC verification process is more efficient than the circularly dependent assumes-guarantees verification methodology. Instead of building models of the system's environment and using them in the verification process, the TLC verification methodology explicitly captures environmental timing properties in the system specification and automatically ensures they are satisfied in the TLC relation. The region-automata-based Timed Logic Conformance System (TLCS) implements TSA parallel composition and a TLC decision procedure. TLCS is used to hierarchically verify the STARI (Self-Timed at Receiver's Input) asynchronous circuit for communicating safely between clock-skewed systems.

TIMED SAFETY AUTOMATA AND LOGIC CONFORMANCE

I. Introduction

"Counting time is not nearly as important as making time count" —Anonymous

1.1 Background

As the practice of specifying, designing, and building computer-based systems evolves, ad-hoc design methodologies are less and less practical. Integrated electronic circuit complexity is growing exponentially. Today, designs with over a million transistors on a single silicon chip are being fabricated, and the technology is doubling the number of transistors possible every 18 months. At the same time more and more functions are being automated and our dependence on electronic technology is increasing exponentially. For example, automotive engineers are working on computer systems that will electronically steer, accelerate, brake—i.e., totally control the vehicle—without a mechanical connection to the driver. Unfortunately, the ability to verify such complex systems has not kept pace with the ability to fabricate them. The risk to life and safety imposed by error-prone computerized systems in military, transportation, industrial, and household control applications is unacceptable. Building custom systems from the ground up is also very expensive and time consuming. Consequently, computer scientists and computer engineers incorporate math-based engineering discipline into the process of specifying, designing, building, and verifying these systems hierarchically by using and reusing mathematical models. This math-based computer engineering discipline is generally called "Formal Methods" (LM91, HJ95, CW96).

Formal methods practitioners write formal specifications and prove that the models of the systems they create satisfy those specifications. Then, they use the model to build the physical system—usually by a combination of automatic (i.e., computerized) and manual transformations. Hopefully, the physical system has the same properties that were proven about the model it was derived from. In order to improve the correspondence between the model and the physical system, more accurate models and more automated transformation processes are always in order. Formal methods have been used successfully to verify the function (i.e., the logical or mathematical relationship between a system's inputs and outputs) of some relatively complex systems (Rus95, CW96).

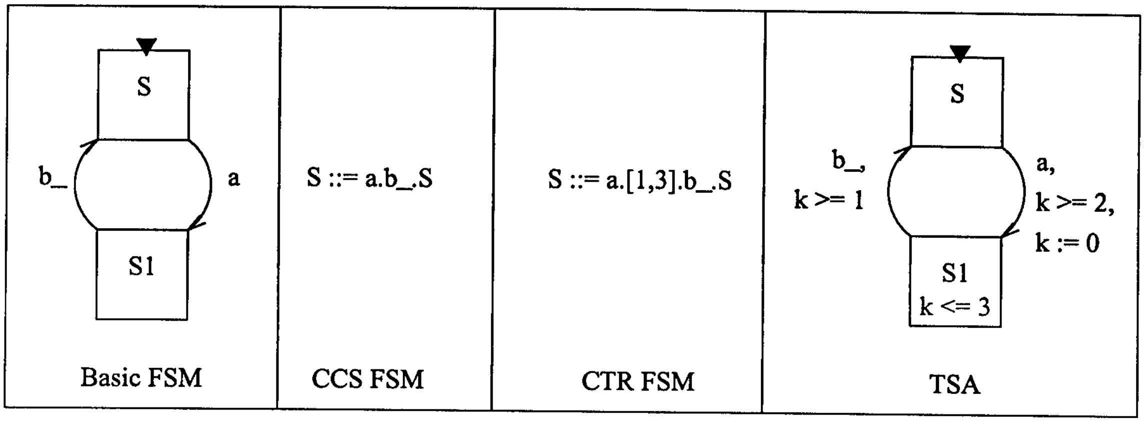
In addition to verifying the systems function, one of the most demanding areas in formal methods is specifying the system's timing requirements, modeling the timing of the behavior, and deciding whether the system satisfies the timing requirements (CW96). For example, if the terrain following control system of an aircraft cannot recognize a mountain looming in the foreground and turn the aircraft or increase its altitude in time to avoid hitting the mountain, then it does not matter that the control system is functionally correct. For over 15 years, the timing problem has occupied the interests of theoretical computer scientists but with relatively little pragmatic application to real-world size problems, especially with regard to a continuous rather than discrete model of time.

Prior to 1991, this work was primarily concerned with temporal logic relationships between system events (All83, All84, Das85, JM86, Lad86, HP87, Tsa87, GF88, BGS89, Jah89, GF90, LA90, MC90). Standard temporal logic does not quantitatively relate events to each other; rather it qualitatively describes the relationship between two events; e.g., given events a and b, a precedes b or b precedes a are two possibilities. Temporal logic also extends the relationships with quantifiers; e.g., event a sometimes (or always) proceeds event b. From 1991 to 1993, work proceeded to specify quantitative timing relationships between events, but it was limited to computing integral timing relationships; e.g. b occurs between one and three time units after a might express either a continuous or discrete time relationship (AR91, Dan92, GI91, RR92, Mol91, Cer92, CH92, GI92, ACD93, CHLM93, Dav93, Jen93, LBGG94). Generally only discrete timing relationships could be computed and verified. From 1993 to the present, representing and reasoning about real-valued timing relationships between events has been possible when integers are used to specify time bounds (ACH94, CLK94, HNSY94, HB94, MRM94, Kan95, Hen95, MP95, SY96, TAKB96, ABK$^+$97, LLPY97, EAP98). Most of the work has focused on checking whether or not certain quantitative temporal logic properties hold in a given model, but little work has been done to formally define and compute the timing relationship between two system models.

The aim of this research is to advance the practical specification of and reasoning about the timing properties of the models computer scientists and electrical engineers use to build systems. In particular, this research is targeted at specifying the behavior and timing of a desired electronic circuit and comparing it to timed models of circuit implementations to see if the behavioral and timing properties of the circuit implementation are consistent with the desired circuit. Typically, the desired circuit and implementation circuit models are at different levels of abstraction; i.e., the desired circuit model is much less detailed than the implementation circuit model. This research uses models with binary functional domains (i.e., voltage levels are either true or false) and a continuous time domain (i.e., time is modeled and measured by real-valued clocks).

One of the most widely used models of behavior with formal semantics is the Finite State Machine (FSM). FSMs are fundamental building blocks for defining and proving properties of languages, protocols, computational complexity, etc. A basic FSM is a set of states and a set of transitions between those states. For the purpose of modeling and building computer systems, engineers typically associate a meaning with the FSM by labeling its transitions and/or states with names that represent some action or process. The Basic FSM on the left of Figure 1 is an example representing a process or agent that inputs a and outputs b_. Changes in input or output value from true to false or vice versa are called **events**. In this document, events are distinguished by labeling transitions with alpha-numerical input and output names; output names have overbars or terminating underscores to distinguish them from inputs. Various flavors of FSM exist, and logics called **process algebras** have been created to use intuitive and concisely-defined FSMs to model and reason about behavior (Hoa85, Mil80, Mil89). The process algebra Calculus of Communicating

Systems (CCS) FSM in Figure 1 represents the same behavior as the Basic FSM with concise CCS syntax.



Basic FSM

CCS FSM

CTR FSM

TSA

Figure 1. Simple State Machines.

Since today's systems are composed of many different concurrently functioning subcomponents, often without a common reference to time, designers must be able to model and reason about subcomponent interactions in a timed calculus. Naturally, this led to timed variants of process algebras (MT92, Kri92, CH92, Dav93, LBGG94, Cer95). The Calculus of Timed Refinement (CTR) (Cer95) FSM in Figure 1 represents almost the same behavior as the Basic and CCS FSMs, except that it constrains output b_ to occur between 1 and 3 time units after input a.

Untimed process algebras and FSM models have been widely used to build complex real-world systems, but timed process algebras have not been effectively used to define and build real-world systems because of the computational complexity of accurately representing and reasoning about the relationship between time and behavior. Discrete models of time reduce the complexity enough to computationally reason about timed FSMs, but since time passes continuously and not in discrete steps, discrete models sacrifice fidelity. Discrete models only allow events to occur at discrete time intervals with regard to each other—i.e., they are synchronized even when they share no causal dependency. Discrete timing models ignore the temporal independence of events. The CTR FSM in Figure 1 can be considered either a discrete or continuous semantic model. Discrete semantics has a b_ transition either 1, 2, or 3 time units after a; continuous semantics has an infinite number of b_ transitions between 1 and 3 time units after a. In the latter case the CTR FSM is not really a *finite* state machine at all. If integers are used to specify delay bounds, time equivalence classes can finitely represent indistinguishable infinite behaviors, so the term "FSM" will continue to be used to generically refer to the different models of computation in this document.

There are basically two ways to reason about the behavior of FSMs: model checking and equivalence checking. **Model checking** is the process of checking the state space of a single FSM to verify that it satisfies or possesses a given property. Model checking properties are expressed in a modal logic or modal μ-calculus (SS94, ANB95). Modal logic is related to modal μ-calculus in the way that propositional logic is related to predicate logic, except the distinguished modal carrier elements are *no-states* and *all-states* and the distinguished propositional logic carriers are *false* and *true*. If a FSM satisfies a given modal logic or modal μ-calculus property, the FSM "models" the property. An example modal logic expression specifying a property of the FSMs depicted in Figure 1 is $[a]\langle b\_\rangle T$, i.e., after every a, there exists a $b\_$ action leading to some state. Properties like deadlock, livelock, and virtually any temporal relationship between actions of FSM can be specified by μ-calculus or temporal-μ-calculus expressions and model checked against FSM. Relatively efficient continuous-time model checking algorithms have been developed and implemented to support timed model checking (FP93, CGL$^+$94, SS95, CL96b, CL96a, LLPY97).

Equivalence checking is the process of comparing two FSMs to each other, and determining if the two machines are similar in some sense. There are various ways to unambiguously define equivalence relationships between FSMs. Some equivalence relationships preserve behavioral properties, and some do not. R. J. van Glabbeek's work is a comprehensive discussion about the relative strength of different equivalence relations between FSMs and the properties they preserve (van90). Some equivalence relationships are simply too strong to be of practical use; others are too weak to preserve some important properties. Typically the equivalence relations strong enough to preserve all properties do not give designers enough freedom of implementation to design efficient systems. Since designers cannot feasibly write down all the formulas necessary to specify all of the important relationships between input and output, typical formal specifications consist of both properties and an abstract model of the desired behavior. Developing an implementation that satisfies the specified properties, and which is "equivalent to" or "implements" the function described by the abstract model is the designer's problem. Therefore, theoretically sound formal methods tools for both model and equivalence checking are important (Pnu98).

Perhaps the first reason timed process algebras have not been effectively used to define and build real-world systems is that timed process algebras add considerable complexity to the untimed logics they are derived from; this makes it hard for humans to intuitively understand the properties and the models. Perhaps the second reason is that timed process algebras are not expressive enough to specify the kinds of timing requirements and properties inherent in today's concurrent systems—especially asynchronous[1] concurrent systems. The fundamental limitation has been the fact that timed process algebras typically only allow one to express integral time passing and then only between two successive events. To be truly useful, one must be able to specify the timing relationship between any two events in a system, and model behavior that occurs over a continuum, not just in discrete integral time steps.

For these reasons, since the early 90's, there has been a lot of work to formalize real-time behavior using continuous models of time and behavior like Timed Safety Automata (AD94) and implement decision procedures for timed model- and equivalence-checking. A **Timed Safety Automaton** (TSA) is a FSM that is augmented with real-valued clocks, location invariant clock predicates, and transition guard clock predicates. The clocks and predicates support intuitive specification of the time and behavior relationship. TSA are more expressive than timed process algebras because they express time relationships between any two events and also between

[1] Asynchronous system components do not have a common clock.

events and states in the model. This expressiveness makes them syntactically more complex than the concise notation of a process algebra. For example, the TSA in Figure 1 expresses the same stimulus-response relationship between input a and output b_ as the CTR FSM, but it also relates occurrences of a to each other via a stimulus-stimulus relationship; i.e., a actions are at least 2 time units apart. TSA can specify any relationship between actions of a system because clocks (like k in this example) can be reset and referenced arbitrarily.

Underlying the intuition of TSA is a semantically precise uncountable-state state machine. In order to reason about the behavior of such a machine in a computer, behavior-distinguishing subintervals of multiple real-valued TSA clock times are symbolically represented in another FSM called a Region Automaton (RA). For systems with multiple clocks, the different possible combinations of subintervals are called regions. For an n-clock TSA, regions are subsets of $\mathbb{R}^n$. Unfortunately, RA suffer from state explosion for systems with more than a few clocks. The requirement that all clocks advance at the same rate and the data structures necessary to correctly maintain the relationships between the clocks causes this state explosion.

Limiting the state-explosion problem in RA-based model-checking and equivalence-checking algorithms has been of considerable interest in the past few years. The primary means of controlling the state-explosion problem for model checking has been to limit the state space explored to only that necessary to verify the property. But, since equivalence checking generally involves comparing all of the reachable states of the models, it has been limited to those problems involving a handful of clocks and very simple models, or to quite loose definitions of equivalence and extra proof obligations (e.g., the assumes-guarantees proof requirements of Berkeley's COSPAN system (TB97)) defining when users can rely on the loose equivalence relations.

1.2 Problem Statement

In the big picture, the most abstract problem statement is, "Incorporate timeliness into system requirements and models and then design reliable constraint satisfying systems." This certainly includes this research, but it more accurately describes the research efforts of hundreds of computer scientists and electrical engineers over the past 15 years. The following research objectives narrow the scope:

1. Adopt or create a simple modeling formalism rich enough to express discrete-valued behavioral properties and timeliness requirements of digital circuits while modeling continuous time.

2. Canonically define how to model digital circuit components and specify required behaviors and timing using the modeling formalism.

3. Formally define a *practical* relationship that expresses when one model satisfies the timing and behavioral requirements of another. Prove that the relation has the necessary mathematic properties for meaningful verification.

4. Write a tractable computational procedure that calculates when the relation holds between two models.

5. Demonstrate the utility of the relation on benchmark digital circuit design problems.

6. Define a verification methodology for using the relation to efficiently and hierarchically verify larger systems.

1.3 Organization

The remainder of this document discusses how the research objectives are met. Chapter II reviews more formally how state machines are used by others to model and reason about behavior. It describes the foundation for this work and sets the stage for seeing and understanding the contribution of this research. It does so by defining and critiquing the syntax, semantics, and relationships for several example formalisms. Chapter II discuss the process algebra CCS in considerable detail and defines both strong and weak notions of "equivalence" between CCS automata. It introduces and explains Ken Stevens' more practical untimed Logic Conformance relation, and the timed formalism Timed CCS. Finally, it defines and explains the assumes-guarantees-based verification methodology for timed processes as implemented in the COSPAN tools from U.C. at Berkeley.

Chapter III defines the syntax and semantics of the modeling formalism used in this research. The Timed Safety Automata (TSA) modeling formalism is successfully used by others for efficient model checking, and it is extended to support this research. Chapter III defines an induced dense-time semantic model for reasoning about TSA behavior called Dense Labeled Transition System (DLTS). It defines how to generate one TSA from another one by restricting, hiding, or renaming its actions. Finally, Chapter III defines how to compose TSA in parallel to make larger and more complex circuit models.

Chapter IV defines the weak "equivalence" relationship called Timed Logic Conformance (TLC). TLC is a timed version of Ken Stevens' Logic Conformance relation. TLC is actually a partial order over the state space of DLTS. This definition introduces abstractions that allow temporal, structural, and behavioral differences between the compared systems in a practical way. Chapter IV establishes that TLC has the necessary mathematic properties to be a useful relationship for efficiently proving when one model is "equivalent to" or "implements" another.

Chapter V describes a finite representation of DLTS that the Timed Logic Conformance System (TLCS) uses to decide whether or not two TSA satisfy the TLC "equivalence" relationship. It describes the TLCS rules and procedures that efficiently implement the TLC decision procedure, and concludes with a description of the TLCS TSA input format, TLCS TSA parallel composition, and TLCS user interface.

Once the TSA and TLC definitions and proofs have been completed in Chapters III and IV, and the TLCS system is described in Chapter V, Chapter VI demonstrates how TLCS can be used to compare models of electronic circuits in a practical way. It demonstrates TLCS's utility on several examples, and it defines canonical modeling practices that increase the fidelity of the circuit models. Chapter VI compares TLCS verification results with others in the literature and it concludes with a summary of the benefits of the TLC methodology and tools.

Finally, Chapter VII summarizes this research, enumerates its contributions, and outlines future work.

II. Existing Models and Relationships

This chapter prepares the reader to appreciate and understand the novel approach to the problem of modeling the behavior and timing of a desired electronic circuit and verifying its consistency with circuit implementation models in the subsequent chapters. This chapter reviews and explains several example formalisms for modeling and reasoning about the "equivalent" behavior of concurrent systems. For each formalism, it defines the syntax of the model, model semantics, and relationships between models.

Starting with untimed process algebraic models and mathematical relationships between these models, Section 2.1 shows that these models are not expressive enough to capture the relationship between time passing and action. It explains how equivalence relations between processes are not structurally and behaviorally loose enough to give designers the freedom they need to design efficiently. It gives an example partial order relation that provides a significantly more practical notion of "equivalence" between untimed processes because it safely gives the designer structural and behavioral looseness.

Section 2.2 and describes and defines three representative timed modeling formalisms and several strong and loose "equivalence" relationships between timed models. It reveals some expressiveness problems with these formalisms, and it critiques the verification methodologies that they support.

2.1 Untimed Models and Relationships

A **process algebra** is a mathematical behavior-modeling language with operational semantics; i.e., semantics defined by rules used to evaluate the meaning of sentences in the language. Process algebras are used to describe and reason about the behavior of concurrent systems. Hoare's Communicating Sequential Processes (CSP) (Hoa85) and Milner's Calculus of Communicating Systems (CCS) (Mil80, Mil89) are process algebras. CSP and CCS were both originally conceived in the early 1980's, and they are still widely used because of their firm theoretical foundation and their simplicity.

Generally CSP is a more complex language than CCS and it uses traces (sequences of actions) to compare processes while generally ignoring the effects internal actions can have on trace generation. If two CSP processes can generate the same set of traces, refusals, or failures, they are considered weakly equivalent. CCS and CSP have the same theoretical expressive power; both are Turing complete (Mil89). CCS models are called "agents". Instead of comparing agents by the traces they can generate, CCS agents are distinguished from one another by notions of bisimulation (i.e., comparing the agents on a state-by-state, action-by-action basis to see if they can always simulate one another or not). Since CCS's notion of equivalence is easier to compute and somewhat more general, and the CCS language is simpler, the next section focuses on CCS.

2.1.1 CCS.

The set of CCS agents is denoted $\mathcal{P}$. The complete syntax for defining a basic CCS agent P is summarized in Table 1.

Table 1. CCS Syntax for Agent P.

Symbol	Name
Nil	empty process
Q	constant
$\alpha.P$	prefix
$P_1 + P_2 + \cdots + P_n$	summation (choice)
$P_1 \mid P_2 \mid \cdots \mid P_n$	composition
$P \setminus L$	restriction
$P[f]$	relabeling

The special CCS agent Nil is "deadlocked" and performs no action; 0 and Nil are used synonymously. The set of actions an agent can perform is its **sort.** Nil's sort is $\varnothing$ (the null set). The semantics of CCS agents (e.g., $P, Q \in \mathcal{P}$) are defined by states and action-labeled transitions that move between states. Such transitions are written like $P \xrightarrow{a} Q$ to mean action a occurs on the transition between states P and Q. The predicate $P \xrightarrow{a}$ is true when there exists some Q such that

$P \xrightarrow{a} Q$. Let $\mathcal{A}$ be the set of input labels, and let $\overline{\mathcal{A}}$ be the set of output labels. Overbarred labels like $\overline{a} \in \overline{\mathcal{A}}$ are outputs[1]. Input and output actions are complementary; i.e., $\overline{\overline{a}} = a$. The language of agents $\mathcal{L} \triangleq \mathcal{A} \cup \overline{\mathcal{A}}$ includes both inputs and outputs[2]. The special label $\tau \notin \mathcal{L}$ represents internal action such that $\overline{\tau} = \tau$, and the action set $Act \triangleq \mathcal{L} \cup \{\tau\}$. Greek letters are used to denote actions; e.g., $\alpha \in Act$.

Given these definitions, the nine named rules in Table 2 define the operational semantics for CCS over the labeled transition system $\langle S, Act, \longrightarrow \rangle$ where S is a set of states, Act is the set of transition labels, and the transition relation $\longrightarrow \subseteq S \times Act \times S$. The rules are precisely semanticized using mathematical arguments of the form:

$$\frac{hypothesis}{conclusion}(condition)$$

Table 2. CCS Operational Semantic Rules.

Act $\dfrac{}{\alpha.E \xrightarrow{\alpha} E}$	**Con** $\dfrac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}(A \triangleq P)$
Sum1 $\dfrac{E \xrightarrow{\alpha} E'}{E+F \xrightarrow{\alpha} E'}$	**Sum2** $\dfrac{F \xrightarrow{\alpha} F'}{E+F \xrightarrow{\alpha} F'}$
Com1 $\dfrac{E \xrightarrow{\alpha} E'}{E\|F \xrightarrow{\alpha} E'\|F}$	**Com2** $\dfrac{F \xrightarrow{\alpha} F'}{E\|F \xrightarrow{\alpha} E\|F'}$
Com3 $\dfrac{E \xrightarrow{\alpha} E' \wedge F \xrightarrow{\overline{\alpha}} F'}{E\|F \xrightarrow{\tau} E'\|F'}$	
Res $\dfrac{E \xrightarrow{\alpha} E'}{E \backslash L \xrightarrow{\alpha} E' \backslash L}(\alpha, \overline{\alpha} \notin L)$	**Rel** $\dfrac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$

The CCS expression $X \triangleq a.b.X$ defines the CCS agent X that according to rule **Act**, performs the following repeating action sequence: $a.b.X \xrightarrow{a} b.X \xrightarrow{b} a.b.X$. Each period, ".", in X's definition corresponds to an unnamed state where X waits for the environment to supply the next action label. The constant rule **Con** supports referring to processes by symbols like R—even recursively to define non-terminating agents. The summation rules **Sum**$_i$ define nondeterministic

[1] In some contexts, trailing underscores (e.g., $a_{\_}$ also represent outputs).

[2] The symbol $\triangleq$ means "is defined as."

choice; allowing processes to execute one of several operations as defined by the summands. The communication rules **Com**$_i$ define the behavior of processes operating in parallel. Parallel composition is denoted by the "|" operator. According to rules **Com1** and **Com2**, agents in parallel continue to perform their individual actions without affecting each other. Rule **Com3** formalizes how two agents, one outputting and one inputting, cooperate to perform an internal action. Other agents cannot participate in the cooperation because the result is a τ-action. Since agents may both continue to perform their individual actions and also cooperate two-at-a-time, CCS's parallel composition is not synchronous. A synchronous calculus requires that all agents able to cooperate in an action do so and it does not allow any agent to continue to perform common or cooperative actions individually. The restriction rule **Res** deletes actions specified in set $L \subseteq \mathcal{L}$. When used in conjunction with **Com3**, **Res** eliminates the individual actions of the composed agents, but not their cooperative τ-action. For example, the two agents $P \triangleq a.P'$ and $Q \triangleq \overline{a}.Q'$ composed in parallel with a restricted by $(P \mid Q)\backslash\{a\}$ results in the transition $(P \mid Q)\backslash\{a\} \xrightarrow{\tau} (P' \mid Q')\backslash\{a\}$ but no $\xrightarrow{a}$ or $\xrightarrow{\overline{a}}$ actions exist for the restricted composition. Finally, rule **Rel** relabels process actions. The relabeling function (f) is defined by supplying tuples $\langle$new/old$\rangle$ specifying the old label and the new label. For example, the process $X \triangleq a.b.X[\langle tic/a\rangle, \langle toc/b\rangle]$ performs the repeating action sequence $a.b.X[\langle tic/a\rangle, \langle toc/b\rangle] \xrightarrow{tic} b.X[\langle tic/a\rangle, \langle toc/b\rangle] \xrightarrow{toc} a.b.X[\langle tic/a\rangle, \langle toc/b\rangle] \ldots$

The behavior of asynchronous hardware systems is defined and reasoned about by associating the voltage changes on wires with instantaneous binary events. A voltage change from the false voltage value to the true voltage value is represented by an instantaneous transition from 0 to 1 and vice-versa for true to false voltage level changes. Transitions between states are labeled with the name of the wire that voltage changes occur on to define the events. For example, the CCS agent $Inv \triangleq a.\overline{b}.Inv$ defines the behavior of a logical inverter with input a and output $\overline{b}$. It also defines the behavior of a buffer, since outputs can be either 0 or 1 in any state. Every $\overline{b}$ transition toggles the output from 1 to 0 or from 0 to 1. Input and output values can be associated with states by naming the states of the inverter with input and output values; e.g., the CCS agents $Inv01 \triangleq a.\overline{b}.Inv10$

and $Inv10 \triangleq a.\bar{b}.Inv01$ associate state names with values. The next section reveals that the states $Inv01$ and $Inv10$ are both equivalent to state Inv and more efficiently represented and reasoned about in CCS as the single state Inv.

2.1.2 CCS Bisimulations. Milner formalizes two basic notions of equivalence between CCS agents. He calls these equivalences **strong bisimulation** and **weak bisimulation**. Bisimulation can be understood as requiring bi-directional simulation between agents; i.e., whatever actions one agent can do, the other can do, and vice-versa. Strong bisimulation demands that even the internal actions (τ-actions) of each agent be matched exactly; weak bisimulation relaxes this requirement.

Definition 1. CCS strong bisimulation. *A binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a strong bisimulation iff $\langle P, Q \rangle \in \mathcal{R}$ implies for all $\alpha \in Act$,*

1. $\forall\, P'[P \xrightarrow{\alpha} P' \Rightarrow \exists\, Q'[Q \xrightarrow{\alpha} Q' \wedge \langle P', Q' \rangle \in \mathcal{R}]]$

2. $\forall\, Q'[Q \xrightarrow{\alpha} Q' \Rightarrow \exists\, P'[P \xrightarrow{\alpha} P' \wedge \langle P', Q' \rangle \in \mathcal{R}]]$

Definition 2. Strongly bisimilar CCS agents: $P \sim Q$. *CCS agents P and Q are strongly bisimilar iff there exists a strong bisimulation $\mathcal{R}$ such that $\langle P, Q \rangle \in \mathcal{R}$. Writing $P \sim Q$ denotes that P is strongly bisimilar to Q.*

The largest strong bisimulation is

$$\sim \; = \bigcup_{\mathcal{R} \in \wp(\mathcal{P} \times \mathcal{P})} \{\mathcal{R} \mid \mathcal{R} \text{ is a strong bisimulation}\}$$

It contains all smaller strong bisimulations and specifies exactly which strong bisimulation must contain $\langle P, Q \rangle$. Milner proves that $\sim$ is an equivalence relation (Mil89:91).

The state relation $\{\langle Inv10, Inv01 \rangle, \langle Inv01, Inv10 \rangle, \langle \bar{b}.Inv01, \bar{b}.Inv10 \rangle, \langle \bar{b}.Inv10, \bar{b}.Inv01 \rangle\}$ is a strong bisimulation relation between the states of the CCS inverter defined above. Given agent

$Inv \triangleq a.\bar{b}.Inv$, the relation $\{\langle Inv, Inv01\rangle, \langle Inv, Inv10\rangle, \langle \bar{b}.Inv, \bar{b}.Inv01\rangle, \langle \bar{b}.Inv, \bar{b}.Inv10\rangle\}$ is also a strong bisimulation, so the *Inv* definition can be substituted for both *Inv*10 and *Inv*01 definitions.

To define weak bisimulation, two abstractions must first be defined. The first abstraction is the transitive reflexive τ-closure: $P(\xrightarrow{\tau})^*P'$ meaning zero or more τ-actions occur between P and P'. When there are no τ-actions, $P = P'$. The transitive reflexive τ-closure leads to a τ-closure over specified actions, which is a superset of the $\longrightarrow$ transition relation. The transitive closure relation is denoted by the double-barred arrow, $\Longrightarrow$.

Definition 3. CCS τ-closure: $P \overset{\alpha}{\Longrightarrow} P'$.

$$\forall\, \alpha \in Act\ P \overset{\alpha}{\Longrightarrow} P' \triangleq P(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* P'$$

Writing $P \overset{\alpha}{\Longrightarrow} P'$ denotes when there is a transition from P to P' by action α in the τ-closure.

Note that $P \overset{\tau}{\Longrightarrow} P'$ means at least one τ occurs between P and P'. Writing $P \overset{\alpha}{\Longrightarrow}$ means there exists some P' such that $P \overset{\alpha}{\Longrightarrow} P'$.

The second abstraction provides a way to match τ-actions with zero or more τ-actions and visible actions by τ-closure. The abstraction is called τ-abstraction. Hatted action symbols (e.g., $\widehat{\alpha}$) denote τ-abstraction; τ-abstraction is used in conjunction with τ-closure to allow structural differences between CCS agents.

Definition 4. CCS τ-abstraction: $P \overset{\widehat{\alpha}}{\Longrightarrow} P'$.

$$\forall\, \alpha \in Act\ P \overset{\widehat{\alpha}}{\Longrightarrow} P' \triangleq \begin{cases} P(\xrightarrow{\tau})^*P' & (\alpha = \tau) \\ P \overset{\alpha}{\Longrightarrow} P' & (\alpha \neq \tau) \end{cases}$$

CCS τ-abstraction is used on the consequent side of bisimulation-style relation formulas to specify that τ-actions in the antecedent can be matched by zero or more τ-actions in the consequent. In the case that they are matched by zero τ-actions, the agent on the consequent side does not perform any action and stays in the same state. This abstraction allows systems being compared to have zero or more structural differences between them and still be considered "equivalent."

Both Milner and Stevens define τ-closure and τ-abstraction over elements from the set of all sequences of actions from *Act* (this set is denoted *Act**), but for this chapter, the above definitions suffice. Stevens uses sequences from *Act** to formally define trace equivalence and trace conformance, his work is a practical discussion of the differences between labeled transition systems distinguished by the different relations (Ste94:pp.111-136).

The τ-closure and τ-abstraction definitions are the basis for defining the weak bisimulation relation for CCS agents. Weak bisimulation allows agents to have different structure but still be considered equivalent based on the actions that they can perform.

Definition 5. CCS weak bisimulation. *A binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a weak bisimulation iff $\langle P, Q \rangle \in \mathcal{R}$ implies for all $\alpha \in Act$,*

1. $\forall\, P'[P \xrightarrow{\alpha} P' \Rightarrow \exists\, Q'[Q \overset{\widehat{\alpha}}{\Longrightarrow} Q' \wedge \langle P', Q' \rangle \in \mathcal{R}]]$

2. $\forall\, Q'[Q \xrightarrow{\alpha} Q' \Rightarrow \exists\, P'[P \overset{\widehat{\alpha}}{\Longrightarrow} P' \wedge \langle P', Q' \rangle \in \mathcal{R}]]$

Definition 6. Weakly bisimilar CCS agents: $P \approx Q$. *CCS agents P and Q are weakly bisimilar iff there exists a weak bisimulation $\mathcal{R}$ such that $\langle P, Q \rangle \in \mathcal{R}$. Writing $P \approx Q$ denotes When P is weakly bisimilar to Q.*

The largest weak bisimulation is

$$\approx \;=\; \bigcup_{\mathcal{R} \in \wp(\mathcal{P} \times \mathcal{P})} \{\mathcal{R} \mid \mathcal{R} \text{ is a weak bisimulation}\}$$

It contains all smaller weak bisimulations and specifies exactly which weak bisimulation must contain $\langle P, Q \rangle$. Milner shows that $\approx$ is an equivalence relation (Mil89:110).

CCS bisimulation is a **congruence** relation; i.e., preserved in all algebraic contexts. Milner proves this (Mil89:98) by showing

$$P_1 \sim P_2 \Rightarrow \begin{cases} (1)\ \alpha.P_1 \sim \alpha.P_2 \\ (2)\ P_1 + Q \sim P_2 + Q \\ (3)\ P_1 \mid Q \sim P_2 \mid Q \\ (4)\ P_1 \setminus L \sim P_2 \setminus L \\ (5)\ P_1[f] \sim P_2[f] \end{cases}$$

Weak bisimulation is not a congruence because summation does not preserve the weak bisimulation relation; e.g., $b.0 \approx \tau.b.0$ but $a.0 + b.0 \not\approx a.0 + \tau.b.0$. However, for the set of CCS agents with guarded actions—i.e., those where every τ action is preceded by a visible action—weak bisimulation is also a congruence.

2.1.3 Logic Conformance. In an effort to further loosen the relationship between CCS agents and safely give designers more freedom, Stevens defines a bisimulation-style partial order relationship called Logic conformance (Ste94). Unlike Milner's strong and weak bisimulation, Logic Conformance is not symmetric, so the implementation agent (I) must be distinguished from the specification agent (S). Usually implementations are less abstract models than specifications. Ultimately, at the lowest level of abstraction, implementations are models of the design primitives from which systems are constructed. In the case of electronic circuits, design primitives are models of logic gates that abstract the voltage levels of the underlying transistor circuits to either true (1) or false (0) and the changes in value from true to false or false to true occur instantaneously.

Definition 7. Logic Conformance. *A binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a logic conformance iff $\langle I, S \rangle \in \mathcal{R}$ implies for all $\alpha \in Act, \beta \in \overline{\mathcal{A}} \cup \{\tau\}, \gamma \in \mathcal{A}$,*

1. $\forall\, S'[S \xrightarrow{\alpha} S' \Rightarrow \exists\, I'[I \overset{\hat{\alpha}}{\Longrightarrow} I' \wedge \langle I', S' \rangle \in \mathcal{R}]]$

2. $\forall\, I'[I \xrightarrow{\beta} I' \Rightarrow \exists\, S'[S \overset{\widehat{\beta}}{\Longrightarrow} S' \wedge \langle I', S'\rangle \in \mathcal{R}]]$

3. $\forall\, I'[(I \xrightarrow{\gamma} I' \wedge S \xrightarrow{\gamma}) \Rightarrow \exists\, S'[S \overset{\gamma}{\Longrightarrow} S' \wedge \langle I', S'\rangle \in \mathcal{R}]]$

Definition 8. Logically Conformant CCS agents: $I \succeq_l S$. *CCS agents I and S are logically conformant iff there exists a logic conformance $\mathcal{R}$ such that $\langle I, S\rangle \in \mathcal{R}$. Writing $P \succeq_l Q$ denotes when P is logically conformant to Q.*

The largest logic conformance is

$$\succeq_l = \bigcup_{\mathcal{R} \in \wp(\mathcal{P}\times\mathcal{P})} \{\mathcal{R} \mid \mathcal{R} \text{ is a logic conformance}\}$$

It contains all smaller logic conformances and specifies exactly which logic conformance must contain $\langle I, S\rangle$. Stevens proves that $\succeq_l$ is a partial order (Ste94:143). Weak bisimulation is the equivalence relation that exists whenever logic conformance holds in both directions; i.e., $I \succeq_l S \wedge S \succeq_l I \Rightarrow I \approx S$.

The difference between weak bisimulation and logic conformance is the extra conjunct in the implication antecedent of Definition 7 property 3. When $S \xrightarrow{\gamma}$ is false, it does not matter that I has a to-state I' reachable by γ (such to-states are called γ **derivatives**). This means that I may accept inputs that S does not. If these unmatched inputs are in the specification's language, then the implementation accepts them more often than the specification does. In case they are not in the specification's language, then they are truly irrelevant and cannot be required of any implementation that replaces S. In either case, the specification defines the inputs that must be accepted and the behavior that follows them. Therefore, in all contexts where the specification accurately represents the desired behavior, logic conformant implementations behave "the same as" the specification. This freedom allows detailed implementations that accurately model the behaviors of design primitives to be considered satisfactory implementations of much less complex specifications even though there are vast differences in their state spaces. Logic conformance provides more freedom of implementation because implementation behavior in unreachable states under the specification's input constraints is completely unrestricted (Ste94:p.120).

Logic conformance does give designers more freedom of implementation, but it does not necessarily preserve all modal logic or modal μ-calculus properties as bisimulation and weak bisimulation do. For example, given $I \triangleq a.b\_.I + c.d\_.0$ and $S \triangleq a.b\_.S$, $I \succeq_l S$, $I \models \langle c \rangle \langle d\_ \rangle T$ but S does not. The symbol "$\models$" is read "models" and means "satisfies property." Conversely, $S \models \langle a \rangle T \wedge [-a]F$ (S can do an a action and no other) but $I \not\models \langle a \rangle T \wedge [-a]F$. In fact, I has a deadlock ($I \xrightarrow{c} \xrightarrow{d\_} 0$), and S does not deadlock. Giving the designer this much freedom of implementation theoretically requires confirming that specification properties specified by modal logic or modal μ-calculus formulae must be model checked in addition to checking the Logic Conformance relation. From a practical point of view if S completely defines I's input environment then conditions like I's deadlock or $d\_$ output can never be reached and model checking is not required.

2.2 Timed Models and Relationships

Untimed process algebras do not provide any power to specify and reason about the timing of different events. That leads us to examine how timing of actions might be specified and reasoned about like CCS agents. There are many different formalisms, each with different syntax, and some more expressive than others (ACD90, CH92, CL96b, Cer92, CGL93, Cer95, CLK94, Dan92, FH92, GF90, GV95, GSSAL94, Hal92, HJ95, HB94, JM86, JU93, Kan95, Koy92, Kri92, Lad86, LY93, LLPY97, LBGG94, MM91, MRM94, Mol91, Wan90). For each formalism slightly different formal relationships have been defined. Three representative formalisms are examined: first, a simple timed-process-algebra extension of CCS called Timed CCS; second, a more expressive timed-process-algebra-based formalism for the Calculus of timed refinement (CTR) relation; third, a very expressive Mealy-machine formalism called a timed process.

2.2.1 TCCS. Wang's Timed CCS (TCCS) extends Milner's CCS with arbitrary integral or real-valued delays (Wan90). Wang writes $P \xrightarrow{\epsilon(t)} Q$ to mean that after t units of time, P becomes Q, where ϵ stands for idling. Note that $P \xrightarrow{\epsilon(0)} P$. TCCS actions other than idling are instantaneous; i.e., no time passes as an action occurs unless the action is an idling action (ϵ-action). TCCS enjoys the same simple syntax of CCS, but it does not readily support specifying temporal relationships between actions that are separated by other observable actions. Wang defines notions of strong and weak timed bisimulation equivalence between TCCS agents. Definitions for Wang's relations are not included here because they are the same as Milner's with the extension of α ranging over both *Act* and ϵ-actions. Both strongly and weakly bisimilar TCCS agents satisfy all timed modal logic formulas that are true of each other (HLY91, LY93). Of course, this means that no temporal differences between observable events can be distinguished between strong or weakly-bisimilar TCCS agents as long as time passes continuously around τ's (i.e., $P \xrightarrow{\epsilon(t1)} (\xrightarrow{\tau})^* \xrightarrow{\epsilon(t2)} Q \Rightarrow P \overset{\epsilon(t1+t2)}{\Longrightarrow} Q$) in the weak case.

There are four special properties of TCCS agents. By definition, τ-actions occur as soon as they are enabled, resulting in a **maximal progress** assumption, i.e., no TCCS agent P will wait unnecessarily to τ:

$$\exists_{Q \in \mathcal{P}}[P \xrightarrow{\tau} Q \Rightarrow \nexists_{t>0}[P \xrightarrow{\epsilon(t)}]]$$

Maximal progress further distinguishes observable actions $\alpha.P$ from internal actions $\tau.P$ because $\tau.P$ has only one τ-transition, and $\alpha.P$ has a chain of $\epsilon(t)$-transitions because $\alpha.P$ is willing to wait for any t units of time for the environment to offer an $\overline{\alpha}$. The maximal progress assumption turns all cooperative α and $\bar{\alpha}$ action pairs into instantaneous τ-actions even though the α- and $\bar{\alpha}$-capable agents may individually be able to wait for arbitrary amounts of time before acting outside of the parallel composition combining them.

Second, TCCS agents are **determinant**; i.e., idling leads to syntactically identical states:

$$P \xrightarrow{\epsilon(t)} P_1 \wedge P \xrightarrow{\epsilon(t)} P_2 \Rightarrow P1 \equiv P2$$

Third, TCCS agents are **continuous**; i.e., TCCS agents must pass through all intermediate time values:

$$P \xrightarrow{\epsilon(t+u)} P_2 \Leftrightarrow \exists\ P_1[P \xrightarrow{\epsilon(t)} P_1 \wedge P_1 \xrightarrow{\epsilon(u)} P_2]$$

Fourth, TCCS agents are **persistent**; i.e., no agent looses its ability to perform an action it was able to perform originally:

$$P \xrightarrow{\epsilon(t)} P_1 \wedge P \xrightarrow{\alpha} P_2 \Rightarrow \exists\ P_1'[P_1 \xrightarrow{\alpha} P_1']$$

Persistence makes it impossible to specify an upper timing bound in TCCS without introducing a τ-transition that forces the agent into another state in accordance with maximal progress. For example, the agent $S0 \triangleq \epsilon(2).a + \epsilon(3).b.S0 + \epsilon(30).\tau.Nil$ can after $t \in [2,30)$ time units do a, and after $t \in [3,30)$ time units do b, and repeat forever. However, if $t = 30$ time units pass without a or b, then $S0$ is deadlocked.

The semantics of TCCS depend greatly on treating τ and visible actions significantly differently, but there is little intuition behind the semantic leap from $P \xrightarrow{\alpha} P' \wedge Q \xrightarrow{\overline{\alpha}} Q'$ to $P \mid Q \xrightarrow{\tau} P' \mid Q'$. Individually, P and Q can wait forever to perform α and $\overline{\alpha}$, but $P \mid Q$ must immediately perform τ in accordance with maximal progress (Wan90).

Recall that specifying time relationships between two actions that have a third action between them is not directly possible without resorting to describing the relationships via multiple parallel agents. For example, assume α and γ never occur closer than 3 time units from each other, but that β can occur at any time. Some possibilities for specifying this situation are $P.\alpha.\epsilon(3).\gamma.\beta.Q +$

$P.\alpha.\epsilon(3).\beta.\gamma.Q + P.\beta.\alpha.\epsilon(3).\gamma.Q$, and even $P.\alpha.\epsilon(1.75).\beta.\epsilon(1.25).\gamma.Q$, but no one can finitely specify all the possible ways to split up the real-valued 3 in this fashion.

TCCS does preserve timed modal logic and μ-calculus properties, but the semantic gap between visible and τ-transitions, its limited provision for specifying upper time bounds, and its constraints on specifying timing relationships between non-sequential actions tend to stifle its practical application and lead us in search of a higher fidelity and more expressive formalism—like the one discussed in the next section.

2.2.2 Calculus of Timed Refinement (CTR). TCCS relaxes the equivalence relation such that the internal structure of the systems may be quite different for weak-timed-bisimilar agents, but it does not allow the timing of the visible actions of those systems to vary. Timed-bisimulation equivalence is too strong a relation for deciding whether or not one agent can be substituted for another, and most agree that even weak-timed-bisimulation overly restricts the freedom of designers.

This notable attempt to give designers more expressiveness and freedom of implementation focuses on the implementation half of the bisimulation relation. Čerāns' Calculus of Timed Refinement (CTR) relaxes the timing relationship between CCS-like agents such that the implementation's timing is more precise than the specification's (i.e., the timing of implementation actions may be a subset of those allowed for specification actions) (Cer95), and CTR provides a way to express minimum and maximum time passing using time intervals.

2.2.2.1 CTR Agents. The set of all CTR agents is denoted $\mathcal{E}$. For CTR agents $E, F, G \in \mathcal{E}, \alpha \in Act$ the syntax used to define agent E is defined by the grammar:

$$nil \mid F \mid [c,e].F \mid \alpha.F \mid F+G \mid F||G \mid F \setminus L \mid F[f]$$

The expression $[c,e].F$ adds the timing delay before F for $c \in \mathbb{R}^+ = (0,\infty), e \in \mathbb{R}^+ \cup \infty, c \leq e$. The expression $\epsilon(d).E$ is another notation for $[d,d].E$. Time progresses by $\epsilon(d)$ units when $[c,e].E \xrightarrow{\epsilon(d)}$

$[b, e-d].E$, $e \geq d$, and $b = max\{0, c-d\}$. CTR introduces another special internal action $i \notin Act$ where $Act^+ = Act \cup \{i\}$ and $\alpha^+ \in Act^+$ ranges over Act^+. The special internal action i denotes when an agent exits the delay prefix sometime after the lower timing bound has been reached, but before the upper timing bound expires; i.e., $[0, e].E \xrightarrow{i} E$. Once the upper bound of the delay prefix expires, the delay prefix can be exited without i occurring; i.e., $[0, 0].E \xrightarrow{\alpha^+} E'$.

Unfortunately, CTR's operational rules defining the semantics of the expressions are considerably more complex than CCS and TCCS, requiring 21 different rules. Table 3 denotes the 12 rules CTR adds to CCS's 9 rules already defined in Table 2. CCS's Con, Com1, Com2, Res, and Rel rules are extended over $\alpha^+ \in Act^+$, and $\alpha \in Act$ applies to the rules in Table 3 for agents $E, F \in \mathcal{E}$.

Table 3. CTR Operational Semantic Rules.

$\dfrac{}{nil \xrightarrow{\epsilon(d)} nil}$	$\dfrac{}{\alpha.S \xrightarrow{\epsilon(d)} \alpha.S}$	$\dfrac{E \xrightarrow{i} E'}{E+F \xrightarrow{i} E'+F}$	$\dfrac{F \xrightarrow{i} F'}{E+F \xrightarrow{i} E+F'}$	$\dfrac{E \xrightarrow{\epsilon(d)} E' \wedge F \xrightarrow{\epsilon(d)} F'}{E+F \xrightarrow{\epsilon(d)} E'+F'}$
$\dfrac{E \xrightarrow{\epsilon(d)} E' \wedge F \xrightarrow{\epsilon(d)} F'}{E \Vert F \xrightarrow{\epsilon(d)} E' \Vert F'} (Sort(d, E) \cap Sort(d, F) = \varnothing)$				
$\dfrac{E \xrightarrow{\epsilon(d)} E'}{E \backslash L \xrightarrow{\epsilon(d)} E' \backslash L}$	$\dfrac{E \xrightarrow{\epsilon(d)} E'}{E[f]L \xrightarrow{\epsilon(d)} E'[f]}$	$\dfrac{}{[0,e].E \xrightarrow{i} E}$	$\dfrac{E \xrightarrow{\alpha^+} E'}{[0,0].E \xrightarrow{\alpha^+} E'}$	$\dfrac{E \xrightarrow{\epsilon(d)} E'}{[c,e].E \xrightarrow{\epsilon(d+e)} E'}$
$\dfrac{}{[c,e].E \xrightarrow{\epsilon(d)} [b,e-d]E} (e \geq d \wedge b = max\{0, c-d\})$				

CTR incorporates Wang's notion of maximal progress, so no further delay is possible when a τ-transition is enabled (Cer95:p519). CTR abstracts structural (τ) and time-passing (i) differences between transition relations by letting $\xrightarrow{i}_i \triangleq (\xrightarrow{i})^*$ and $E \xrightarrow{\alpha}_i E'$ denote $E \xrightarrow{i}_i \xrightarrow{\alpha} \xrightarrow{i}_i E'$, and letting also $\xrightarrow{\tau}_\tau \triangleq (\xrightarrow{\tau})^*$ and $\xrightarrow{\alpha}_\tau$ denote $\xrightarrow{\tau}_\tau \xrightarrow{\alpha} \xrightarrow{\tau}_\tau$, as well as $\xrightarrow{\tau}_{i,\tau} = \xrightarrow{i}_{i,\tau} \triangleq (\xrightarrow{\tau} \cup \xrightarrow{i})^*$ and $\xrightarrow{\alpha}_{i,\tau} \triangleq \xrightarrow{\tau}_{i,\tau} \xrightarrow{\alpha} \xrightarrow{\tau}_{i,\tau}$.

CTR defines matching delay transitions from delay prefixes ($d \in \mathbb{R}^+$) to a CTR agent $E \in \mathcal{E}$ by functions $f_-, f_+ : [0, d] \to \mathcal{E}$ such that:

- $f_-(0) = E$

- And
  - $\forall\, d' \in [0,d][f_-(d') \xrightarrow{\tau}_{i,\tau} f_+(d')]$
  - $\exists\, d_0, d_1, \ldots, d_k \in [0,d][d_0 = 0 \wedge d_k = d \wedge \forall\, 0 \leq i < k[d_{i+1} > d_i]]$ and
    * $\forall\, d' \in [0,d] \setminus \{d_0, d_1, \ldots, d_k\}[f_-(d') = f_+(d')]$
    * $\forall\;\; 0 \leq j < k, d' \in (d_j, d_{j+1}][f_+(d_j) \overset{\epsilon(d'-d_j)}{\longrightarrow} f_-(d')]$

Under these conditions the pair of functions $f = \langle f_-, f_+ \rangle$ is an $\langle E, d \rangle$-trace. The set of $\langle E, d \rangle$-traces is denoted by $E \xrightarrow{\epsilon(d)}_{i,\tau}$, and $E \xrightarrow{\epsilon(d)} \subseteq E \xrightarrow{\epsilon(d)}_{\tau} \subseteq E \xrightarrow{\epsilon(d)}_{i,\tau}$ are the sets of $\langle E, d \rangle$-traces not involving any internal transitions and not involving $\xrightarrow{i}$ respectively.

For two CTR agents I and S, $f \in I \xrightarrow{\epsilon(d)}_{i,\tau}$ and $g \in S \overset{\epsilon(d)_{i,\tau}}{\longrightarrow}$, and the relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ the predicate $\langle f, g \rangle \in \mathcal{R}^t$ is true if

$$\forall\, d' \in [0,d][\langle f_+(d'), g_+(d') \rangle \in \mathcal{R} \wedge \langle f_-(d'), g_-(d') \rangle \in \mathcal{R}]$$

The predicate $\langle f, g \rangle \in \mathcal{R}^t$ is used to require that S be able to match I's observable behavior continuously while I passes time d.

The predicate $E \xrightarrow{\tau^\infty}_i$ denotes an infinite chain of $\xrightarrow{\tau}_i$ transitions starting at E: $E \xrightarrow{\tau}_i E_1 \xrightarrow{\tau}_i E_2 \xrightarrow{\tau}_i \cdots$.

2.2.2.2 CTR Relations.

Definition 9. CTR Refinement Relation. *A relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ is a CTR refinement relation iff*

$$\forall\, I \mathcal{R} S[$$

$$I \xrightarrow{\alpha^+} I' \;\;\Rightarrow\;\; S \xrightarrow{\alpha^+}_{i,\tau} S' \wedge I' \mathcal{R} S' \wedge \tag{1}$$

$$S \xrightarrow{\alpha} S' \;\;\Rightarrow\;\; I \xrightarrow{\alpha}_{\tau} I' \wedge I' \mathcal{R} S' \wedge \tag{2}$$

$$f \in I \xrightarrow{\epsilon(d)} \quad \Rightarrow \quad g \in S \xrightarrow{\epsilon(d)}_{i,\tau} \wedge \langle f, g \rangle \in \mathcal{R}^t \wedge \tag{3}$$

$$I \xrightarrow{\tau^\infty}_i \quad \Rightarrow \quad S \xrightarrow{\tau^\infty}_i] \tag{4}$$

Note that CTR asymmetrically specifies the requirements for action matching between specification S and implementation I. Formula 1 requires that all $I \xrightarrow{i}$ internal actions exiting delay prefixes be matched by the specification S, but not conversely in Formula 2. In the same formulas, the specification is given more flexibility for matching the implementation, closing its relation over i as well as τ. Further, only implementation delays actions $\epsilon(d)$ need be matched according to Formula 3. There is nothing requiring the implementation to match specification delays actions. Formula 4 ensures that internal divergence by the implementation is also possible by the specification.

Definition 10. CTR Refinement: *$I \trianglelefteq S$. CTR agent I refines CTR agent S iff there exists a CTR refinement relation $\mathcal{R}$ such that $\langle I, S \rangle \in \mathcal{R}$. Writing $I \trianglelefteq S$ denotes when I refines S.*

2.2.2.3 CTR Summary. CTR preserves timed modal-logic and μ-calculus properties. In fact, it implies TCCS weak-timed bisimulation when limited to the subset of timing relationships expressed by TCCS agents (Cer95:p.525).

Ultimately for all its complexity, CTR's only real benefit over TCCS is the fact that CTR relaxes the time relationship between actions of the two systems. In CTR, the agent $I ::= a.[1,3].b\_.I$ refines $S ::= a.[0,4].b\_.S$ but they are not TCCS weakly bisimilar because S can do $b\_$ immediately after a, but I cannot.

CTR's delay prefixes are a clean way to specify upper time bounds that cannot easily be specified with TCCS agents. CTR is still limited to specifying timing relationships between sequential actions (e.g., separating occurrences of a in the TSA from Figure 1 by two or more time units is not possible in CTR). Also, since CTR maintains TCCS's maximal progress property, CTR does not close the semantic gap between visible and τ-transitions. CTR refines the timing of both input and output actions such that they are both "more-precise" in the implementation than the specification. A refinement relation that allows an implementation not to accept every input that the specification does is not very useful. For these reasons, and because of the computational complexity of algorithms to implement it, CTR is not used extensively to specify, design, and reason about real designs.

2.2.3 Timed Simulation and Assumes-Guarantees Reasoning. The second notable attempt to give designers more freedom of implementation also focuses on the implementation half of the bisimulation relation (TAKB96). These Bell Labs and Berkeley researchers have implemented a system (timed COSPAN) for checking timed simulation relations and for doing hierarchical and compositional verification with assumes-guarantee style proof rules. The primary difference between their modeling formalism and CTR is the computational model.

2.2.3.1 Timed Processes. The computational model of the COSPAN formalism is a Moore machine called a **timed process**, not a CCS, TCCS, or CTR agent. Timed processes are more complicated than TCCS or CTR agents, but they provide the capability to temporally relate actions that do not occur sequentially by declaring and resetting clock variables on transitions and using those clock variables in predicates that determine when actions may occur. The timed process definition follows.

Let X be a finite set of real-valued clock variables. An **X-valuation** Φ assigns a nonnegative real value $\Phi(x)$ to each variable $x \in X$. Let Φ be an X-valuation, and for each real-valued $\delta \geq 0$, $\Phi + \delta$ denotes the X-valuation assigning $\Phi(x) + \delta$ to each variable x, and $\vec{0}$ denotes the X-valuation assigning 0 to every $x \in X$. For $Y \subseteq X$, $\Phi[Y := 0]$ denotes the X-valuation assigning 0 to every $y \in Y$ and $\Phi(x)$ to $x \notin Y$ (a projection). An **X-predicate** φ is a positive Boolean combination of constraints of the form $x \diamond k$ for k an nonnegative integer constant, $x \in X$ a variable, and $\diamond \in \{\leq, \geq, =\}$. Writing $\Phi \models \varphi$ denotes that Φ satisfies the X-predicate φ.

For P a finite set of variables, each ranging over a finite domain, a **P-valuation** $\vec{f}$ is an assignment of values to variables in P. For $\vec{f}$ and $Q \subseteq P$, $\vec{f}(Q)$ denotes the Q-valuation restricting $\vec{f}$ to the variables in Q. A P-event is a pair $\langle \vec{f}, \vec{f}' \rangle$ denoting the old $(\vec{f})$ and new $(\vec{f}')$ values of variables in P. A **P-predicate** χ is a subset of P-events. For example the P-predicate $p' \neq p$ is the set of all P-events $\langle \vec{f}, \vec{f}' \rangle$ such that $\forall p \in P[\vec{f}'(p) \neq \vec{f}(p)]$. To ensure all variable assignments stay the same, the predicate $Stutter(P)$ is defined as:

$$Stutter(P) = \bigwedge_{p \in P} p' = p$$

Definition 11. Timed Process. *Let $\mathcal{TP}$ be the set of all timed processes. A timed process $A \in \mathcal{TP}$ is an eight-tuple $\langle S, S_0, X, O, I, \alpha, \mu, E \rangle$ such that*

- *S is a finite non-empty set of locations.*
- *S_0 is the non-empty set of initial locations.*
- *X is the finite set of real-valued clock variables.*
- *O and I are finite sets of output and input variables, each ranging over a finite type, $I \cap O = \varnothing$.*
- *α is the* **invariant function** *assigning the X-predicate $\alpha(s)$ to each location $s \in S$.*
- *μ is the* **output function** *assigning $\mu(s)$ to each location $s \in S$.*
- *E is the finite set of edges. Each edge $e \in E$ is a 5-tuple $\langle s, t, \varphi, \chi, Y \rangle$ with source and destination locations s and t, clock predicate φ, input predicate χ, and the set of clocks $Y \subseteq X$ to be reset. Two modeling constraints are:*

  1. *$\forall\ s \in S[\langle s, s, true, stutter(I), \varnothing \rangle \in E]$*
  2. *For every pair of locations there is at most one edge between them.*

A **state** σ of A is a pair $\langle s, \Phi \rangle$ containing the location s and the X-valuation $\Phi \in \alpha(s)$, and the set of states is Σ_A. A state $\langle s, \Phi \rangle$ is **initial** if $s \in S_0$ and $\forall\ x \in X[\Phi(x) = 0]$.

Given state $\sigma = \langle s, \Phi \rangle$ of A, and positive time increment δ, A can wait for δ in state σ, written $wait(\sigma, \delta)$, iff $\forall\ 0 \leq \delta' < \delta[(\Phi + \delta') \models \alpha(s)]$. A **timed event** γ of A is a tuple $\langle \delta, \vec{f}, \vec{f}' \rangle$ consisting of a positive real-valued increment δ and the observation event $\langle \vec{f}, \vec{f}' \rangle$. Such an event means that A can wait for δ time and then update its output from $\vec{f}(O)$ to $\vec{f}'(O)$ while the environment is updating the input variables from $\vec{f}(I)$ to $\vec{f}'(I)$. The set of all A timed events is denoted Γ_A.

The timed process A gives a labeled transition system over the state space Σ_A with the labels Γ_A. For states $\sigma = \langle s, \Phi \rangle$ and $\tau = \langle t, \Theta \rangle$ in Σ_A, and a timed event $\gamma = \langle \delta, \vec{f}, \vec{f}' \rangle$ in Γ_A, the transition $\sigma \xrightarrow{\gamma} \tau$ is defined iff $\vec{f}(O) = \mu(s)$, $\vec{f}'(O) = \mu(t)$, $wait(\sigma, \delta)$, and there exists and edge $\langle s, t, \varphi, \chi, Y \rangle$ such that $(\Phi + \delta) \models \varphi$, $\langle \vec{f}, \vec{f}' \rangle \models \chi$, and $\Theta = (\Phi + \delta)[Y := 0]$. Writing $\sigma \xrightarrow{\gamma}$ denotes that $\sigma \xrightarrow{\gamma} \tau$ for some τ.

Timed Processes are closed under stuttering; i.e., let $\gamma = \langle \delta, \vec{f}, \vec{f}' \rangle$, then:

$$\sigma \xrightarrow{\gamma} \tau \Rightarrow \forall\ 0 < \delta' < \delta[\exists\ \sigma', \gamma', \gamma''[\sigma \xrightarrow{\gamma'} \sigma' \xrightarrow{\gamma''} \tau \wedge \gamma' = \langle \delta', \vec{f}, \vec{f} \rangle \wedge \gamma'' = \langle \delta - \delta', \vec{f}, \vec{f}' \rangle]]$$

Figure 2 is an example timed process defining the behavior of an inertial buffer with input i, output o, and delay in $[MinD, MaxD]$. The stutter-closing self loops are omitted in the figure. Every process spends non-zero time in each location, and all transitions are instantaneous. Initial locations are denoted by ▶.

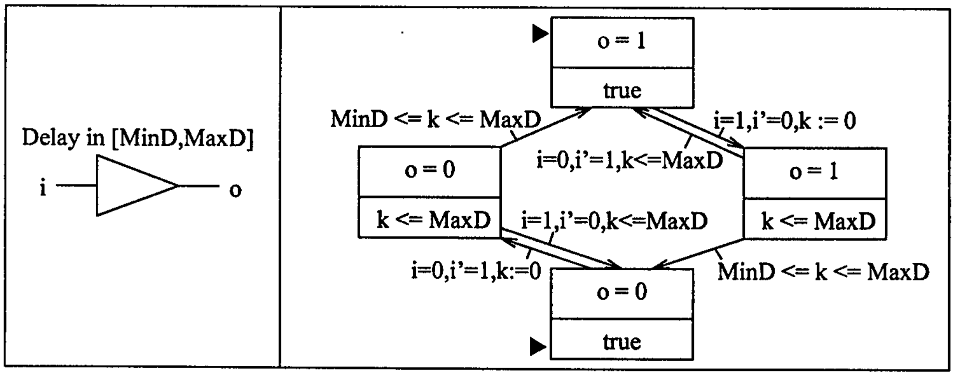


Figure 2. Inertial Buffer Timed Process.

The following definition specifies how to model more complex systems by parallel composing simpler timed process models together. Here, "\" denotes set subtraction.

Definition 12. Timed Process Parallel Composition. *Two timed processes:*

$$\begin{aligned} A &= \langle S^A, S_0^A, X^A, O^A, I^A, \alpha^A, \mu^A, E^A \rangle \\ B &= \langle S^B, S_0^B, X^B, O^B, I^B, \alpha^B, \mu^B, E^B \rangle \end{aligned}$$

can be parallel composed iff $O^A \cap O^B = \varnothing$ *(they share no common output). The parallel composition* $P = A||B$ *is a timed process* $P = \langle S^P, S_0^P, O^P, I^P, \alpha^P, \mu^P, E^P \rangle$ *such that*

$$\begin{aligned} S^P &= S^A \times S^B \\ S_0^P &= S_0^A \times S_0^B \\ X^P &= X^A \cup X^B \\ O^P &= O^A \cup O^B \\ I^P &= (I^A \cup I^B) \setminus O^P \\ \alpha^P &= \forall\, s \in S^A, t \in S^B [\alpha^P(\langle s,t \rangle) = \alpha^A(s) \wedge \alpha^B(t)] \\ \mu^P &= \forall\, s \in S^A, t \in S^B [\mu^P(\langle s,t \rangle) = \mu^A(s) \cup \mu^B(t)] \\ E^P &= \{\langle \langle a,b \rangle, \langle a',b' \rangle, \varphi \wedge \varphi', \chi'', Y \cup Y' \rangle \mid \langle a,a',\varphi,\chi,Y \rangle \in E^A \wedge \langle b,b',\varphi',\chi',Y' \rangle \in E^B \wedge \\ &\quad ((\langle \vec{f}, \vec{f'} \rangle \in \chi'' \iff ((\langle \vec{f} \cup \mu^B(a'), \vec{f'} \cup \mu^B(b') \rangle \models \chi \wedge \langle \vec{f} \cup \mu^A(a), \vec{f'} \cup \mu^A(b) \rangle \models \chi'))\} \end{aligned}$$

The set of edges in the parallel composition consist of those edges where the outputs of each individual process (μ^A, μ^B) satisfy the input predicates of the other process (χ', χ). Timed process parallel composition is commutative and associative.

2.2.3.2 Timed Process Relations. Simulation relations between Timed Processes are defined over **timed event sequences.** A timed event sequence $\overline{\gamma} = [\gamma_0, \gamma_1, \ldots, \gamma_{k-1}]$ is a finite sequence of events $\gamma_i = \langle \delta_i, \vec{f_i}, \vec{f_i'} \rangle$ such that $\forall\ 0 \leq i < k-1[\vec{f}_{i+1} = \vec{f_i'}]$. For such a timed event sequence, define $\Delta_0 = 0$, and $\Delta_i = \Sigma_{j=0}^{i-1}\delta_j$ for $1 \leq i < k$. Each such $\overline{\gamma}$ uniquely defines a function $F_{\overline{\gamma}}$ from the closed interval $[0, \Delta_k]$ to the observations given by $F_{\overline{\gamma}}(t) = \vec{f_i}$ for $t \in [\Delta_i, \Delta_{i+1})$ and $F_{\overline{\gamma}}(\Delta_k) = \vec{f}'_{k-1}$.

A **run** of A on a timed event sequence $\overline{\gamma}$ is a sequence of states $[\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_k], \sigma_i \in \Sigma_A$ such that $\sigma_0 \xrightarrow{\gamma_0} \sigma_1 \xrightarrow{\gamma_1} \sigma_2 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_{k-1}} \sigma_k$. The timed event sequence $\overline{\gamma}$ is called a **trace** of A if there exists a run in A on $\overline{\gamma}$ starting from an initial state and terminating in a state $\sigma_k \in \Sigma_A$. The **timed language** of process A, denoted $\mathcal{L}(A)$, is the set of traces of A.

Consider Two timed processes:

$$\begin{aligned} A &= \langle S^A, S_0^A, X^A, O^A, I^A, \alpha^A, \mu^A, E^A \rangle \\ B &= \langle S^B, S_0^B, X^B, O^B, I^B, \alpha^B, \mu^B, E^B \rangle \end{aligned}$$

A is **comparable** to B iff $O^B \subseteq O^A \wedge I^B \subseteq I^A$; i.e., B's outputs and inputs are subsets of A's.

If A is comparable to B, then **a timed simulation relation** from A to B is a binary relation $\Omega \subseteq \Sigma_A \times \Sigma_B$ among the states of A and B such that

$$\forall\ \langle \sigma, \tau \rangle \in \Omega, \gamma \in \Gamma_A[\sigma \xrightarrow{\gamma} \sigma' \Rightarrow \exists\ \tau' \in \Sigma_B[\tau \xrightarrow{\gamma} \tau' \wedge \langle \sigma', \tau' \rangle \in \Omega]]$$

The timed simulation relation Ω is **initialized** iff $\forall\ \sigma \in S_0^A[\exists\ \tau \in S_0^B[\langle \sigma, \tau \rangle \in \Omega]]$. If A is comparable to B and an initialized timed simulation relation from A to B exists, then A **timed-simulates** B written $A \preceq_S B$.

Let A be comparable to B, then A is said to **timed-implement** B **iff**

$$\forall\, \overline{\gamma}^A \in \mathcal{L}(A)[\exists\, \overline{\gamma}^B \in \mathcal{L}(B)[F_{\overline{\gamma}^A}(I^B \cup O^B) = F_{\overline{\gamma}^B}]]$$

i.e., the traces of the two machines assign the same values to B's input and output variables at all times. Timed implementation is denoted $A \preceq_L B$, and is also referred to as the **language inclusion relation**. The relations $\preceq_S$ and $\preceq_L$ are reflexive and transitive. When $A \preceq_S B$ and $B \preceq_S A$, then A and B are **timed simulation equivalent**, written $A \cong_S B$; $\cong_S$ is an equivalence relation. Similarly, $\cong_L$ is the equivalence relation induced by $\preceq_L$.

Timed simulation is a stronger requirement than timed implementation; i.e., $A \preceq_S B \Rightarrow A \preceq_L B$. The timed simulation relation can be decided in an exponential algorithm when the timing of the two processes are represented by finite equivalence classes, so timed simulation is the relation checked by the COSPAN system to decide when system A can be substituted for system B. If $A \preceq_L B$ then A **refines** B.

Timed process parallel composition preserves both $\preceq_S$ and $\preceq_L$; i.e., $\forall X[A \preceq_S B \Rightarrow A\|X \preceq_S B\|X]$ and $\forall X[A \preceq_L B \Rightarrow A\|X \preceq_L B\|X]$. So $A\|B \preceq_S P\|Q$ if $A \preceq_S P \wedge B \preceq_S Q$; this allows decomposing large verifications into smaller pieces.

2.2.3.3 Assumes-Guarantees Verification. Timed Simulation verification involves generating timed process models of the environment and composing them with timed process models of the desired system and reasoning about their behavior together. Consequently, the behavior of the system depends on the behavior of the environment which depends on the behavior of the system in a circular fashion. To break this circular chain of dependency an **assumes-guarantees** proof methodology is adopted. The methodology depends on the fact that a composed process is an implementation of each of its components (i.e., $A\|B \preceq_S A$). This fact is used to make assumptions about the rest of the system's behavior when trying to determine if a component (the one being designed) satisfies a more abstract specification of its own behavior. Composed timed processes must be **nonblocking** for a consistent assumes-guarantees proof methodology.

A timed process $A = \langle S, S_0, X, O, I, \alpha, \mu, E \rangle$ is **nonblocking iff**

$$\forall\, \sigma \in \Sigma_A, \langle \delta, \vec{f}, \vec{f}' \rangle \in \Gamma_A[\sigma \xrightarrow{\langle \delta, \vec{f}, \vec{f}' \rangle} \Rightarrow \forall\, \langle \delta, \vec{g}, \vec{g}' \rangle \in \Gamma_A[\vec{g}(O) = \vec{f}(O) \wedge \vec{g}'(O) = \vec{f}'(O) \Rightarrow \sigma \xrightarrow{\langle \delta, \vec{g}, \vec{g}' \rangle}]]$$

Intuitively, this means that nonblocking processes should be able to generate a trace regardless of the sequence of input events. In this case, if after δ time passes A updates its output from $\vec{f}(O)$ to $\vec{f}'(O)$ and at the same time the environment updates the inputs from $\vec{g}(I)$ to $\vec{g}'(I)$ there must be an edge in E from σ to a state for $\langle \delta, \vec{g}, \vec{g}' \rangle$'s input condition with consistent output. Hence the updating of O is independent from I, and there must be an edge to a state allowing them to be independent. Generally, nonblocking requires defining edges for all possible input conditions from all possible states (GSSAL94).

Definition 13. Assumes-Guarantees Rule. *Given nonblocking timed processes* $A, B, C, D \in \mathcal{TP}$*:*

$$((A||D \preceq_L C) \wedge (C||B \preceq_L D)) \Rightarrow A||B \preceq_L C||D$$

The assumes-guarantees rule says that proving A is a refinement of C assuming that the environment behaves like D and proving that B is a refinement of D assuming that the environment behaves like C establishes that $A||B \preceq_L C||D$. The assumes-guarantees rule does not hold for blocking processes, and it does not hold if $\preceq_S$ replaces $\preceq_L$. The rule also fails if time predicates defining open sets are used; i.e., strict inequalities like $k < 5$ or $k > 5$ cannot be used in state invariants $\alpha(\sigma)$ or edge clock predicates φ.

Timed simulation verification methodology using the assumes-guarantee rule is expensive in practice requiring one or more processes to model the environment and at least $2n$ verifications and $3n$ timed process models for n-process compositions ($n > 2$). For example, given the structure of the 3-process system (including the environment processes) shown in Figure 3, where there are two timed process models for each of the three components, one concrete (e.g., X_c) and one abstract (e.g., X_a), the problem is to decide whether or not the composition of the concrete models refine the abstract composition. Three full-size verifications must be done.

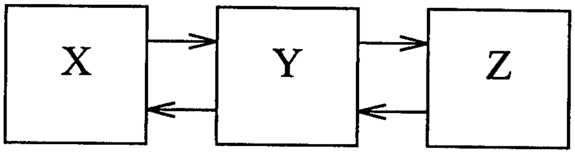


Figure 3. Assumes-Guarantees Example.

In this case, to conclude $X_c||Y_c||Z_c \preceq_L X_a||Y_a||Z_a$, the verifications $X_c||Y_a||Z_a \preceq_L X_a||Y_a||Z_a$, $X_a||Y_c||Z_a \preceq_L X_a||Y_a||Z_a$, and $X_a||Y_a||Z_c \preceq_L X_a||Y_a||Z_a$ must all be successful. In practice, the combined state space of both the concrete and abstract compositions is too large and the verification takes too long, so a single abstract process is developed to represent the behavior of all of the other systems in the composition (i.e., the environment from the perspective of any one of the abstract processes in the composition), and it is used to reduce the state space of the verification problem. For example, process models XY, YZ, and XZ modeling the environments $X_a||Y_a$, $Y_a||Z_a$, and $X_a||Z_a$ are needed, and the verifications $X_c||YZ \preceq_L X_a||YZ$, and $Y_c||XZ \preceq_L Y_a||XZ$, and $Z_c||XY \preceq_L Z_a||XY$ are required. These verifications are valid only if the environmental abstractions are correct, so $X_a||Y_a \preceq_L XY$, $Y_a||Z_a \preceq_L YZ$, and $X_a||Z_a \preceq_L XZ$ must also be verified. This requires total of 6 verifications and 9 different timed process models to verify a 3-element composition.

When any model changes, every verification involving it must be redone. If an abstract model changes then at least n verifications must be redone, but if a concrete model changes, only 1 verification need be redone. Unfortunately, the most difficult models to construct are the abstraction models, and the assumes-guarantees methodology requires more abstract models than concrete models. Clearly, when models are subject to change frequently, as they are in most design and verification projects, assumes-guarantees verification is a significant effort.

Additionally, the timed COSPAN tool does not calculate the simulation relation; rather, the user must input a map from location to location and COSPAN checks to see if the mapping is a simulation relation. Commenting on this, the COSPAN users desire a capability for automatically generating the simulation relation or checking if there is an initialized one without generating it (TAKB96). They also describe the process of generating accurate abstract models as an iterative process; hence the $2n$ verifications were redone many times, and each time they had to supply the appropriate simulation relation.

2.2.3.4 Timed Simulation and Assumes-Guarantees Summary. The timed process formalism is more powerful than TCCS agents and CTR processes because it resolves the problem of expressing timing relationships between any two actions by resetting clocks and referencing them in clock predicates. Timed process definitions are quite complicated because they use state functions to define outputs and invariants, and they use sequences of events to define process semantics.

The nonblocking property required for the assumes-guarantees rule also makes it difficult to create simple timed process models. In order to be consistent in all verification contexts, the model must define behavior for all inputs for all states for all times to ensure that the nonblocking property holds. In contrast to CTR refinement, this means that input behaviors are not refined at all; they must be continuously specified. This is like trying to formally define and derive programs and subprograms using preconditions, postconditions, and Dijkstra calculus, but the only precondition allowed is the weakest of all—i.e., **true** (Dro89).

Clearly, to simplify the modeling burden, and to distribute the burden of verification rationally, something besides weakening the input constraints in implementations (as in CTR refinement) or specifying all inputs in all states for all time (as in the assumes-guarantees methodology) must be done. A way to factor the timing properties of the environment into the verification process without having to build many different models of the environment from each component's perspective must be found.

2.3 Summary

This chapter introduced and defined several example formalisms for modeling and reasoning about the "equivalent" behavior of concurrent systems. Some formalisms cannot express the relationship between time and behavior at all, while others that can are very complicated and hard to define. Some equivalence relationships between models in those formalisms are strong and preserve properties, but they do not give the designers enough freedom to design efficiently. Other weaker relationships give designers structural and behavioral freedom to design and specify more efficient implementations, but they do not necessarily preserve all possible properties.

Untimed CCS agents are not expressive enough to capture the relationship between time passing and action, and the equivalence relations bisimulation and weak-bisimulation between agents are not loose enough to give designers the freedom they need to design efficiently.

The untimed partial order relation Logic Conformance provides a significantly more practical notion of "equivalence" between untimed CCS agents but it does not generally preserve modal-logic or μ-calculus properties.

The three representative timed modeling formalisms Timed Calculus of Communicating Systems (TCCS), Calculus of Timed Refinement (CTR), and timed processes have some expressiveness problems:

- Upper and lower time bounds (bi-bounded delays) are difficult to define in TCCS.
- The maximal-progress semantic leap from two processes waiting individually to perform their actions to cooperating processes that cannot wait to perform their cooperative actions is a fidelity problem for both TCCS and CTR.
- General temporal relationships between actions that do not sequentially follow each other are impossible to express in TCCS and CTR.

- Timed processes support expressing general temporal relationships between actions, but they are quite complicated because they use state functions to define outputs and invariants, and sequences of events to define process semantics.

This chapter referenced the timed "equivalence" relationships timed bisimulation and weak timed bisimulation for TCCS agents. It defined the timed "equivalence" relationships: CTR-refinement for CTR agents; and timed-simulation and timed implementation between timed processes.

It also described, defined, and critiqued the assumes-guarantees verification methodology used with the most expressive formalism—timed processes. At-best the most expressive and practical modeling and verification task is formidable because of the circular dependencies between the environment and the system inherent in the assumes-guarantees verification methodology. The iterative nature of generating accurate abstractions and using them to simplify verification computations forces one to always consider the entire system in the verification or reaccomplish many "equivalence" checks to verify the verification. And, the most advanced tool, COSPAN, requires the user to supply an untimed simulation relation between the states of the systems being compared instead of directly computing it.

Designers need a "simple" modeling formalism that powerfully expresses the relationship between behavior and time. There must be a way to factor the timing properties of the environment into the verification process without building many different models of the environment and using them to "verify the verification." Designers need a formal mathematical relationship that accurately defines an acceptable implementation relation between models in a practical way that can be computed efficiently without a lot of user input required.

III. Timed Safety Automata

The first step towards addressing the shortcomings revealed in Chapter II is choosing a simple and expressive formalism as a timed model for concurrent systems. This chapter formally defines the Timed Safety Automata (TSA) formalism used in this research to specify and model timed system behavior. This TSA model is simpler than COSPAN timed processes, but at the same time it suffers none of the expressiveness problems associated with untimed process algebras, TCCS and CTR. The TSA formalism has been extensively studied and has been widely used. For a formal exposition of TSA expressiveness and computational complexity see (AD94). In this research, a flavor of TSA with both location and transition predicates and action-labeled transitions is used to model digital circuits.

The chapter includes basic TSA definitions, TSA semantics, TSA modification rules, and TSA parallel composition rules. The following TSA definition is based on Sokolsky's (SS95). It supports a dense-time model of time with the non-negative real numbers $\mathbb{R} \triangleq [0, \infty)$, and time constants from the non-negative integers $\mathbb{Z} \triangleq \{0, 1, 2, ...\}$.

3.1 Basic TSA Definitions

Definition 14. TSA. *Let* $\mathbb{T}$ *denote the set of TSA. Given*

- **clock***: A clock* ξ *is an* $\mathbb{R}$*-valued variable. Let* C *be the set of clock variables.*

- **clock constraint***: A clock constraint is an expression of the form* $\xi\ R\ c$ *where* $\xi \in C$, $R \in \{\leq, \geq, <, >\}$*, and* $c \in \mathbb{Z}$.

- **clock assignment***: Given the ordered set* $\Xi = (\xi_1, \xi_2, \ldots, \xi_n) \subseteq C$*, a clock assignment* $\vec{\pi} = (x_1, ..., x_n) \in \mathbb{R}^n$ *is an instantiation of* Ξ.

- **idling***:* $\vec{\pi} + d \triangleq (x_1 + d, ..., x_n + d)\ d \in \mathbb{R}$.

- **clock reset**: *Given $\eta \subseteq \Xi$, a clock reset $\vec{\pi}[\eta := 0]$ projects a clock assignment $\vec{\pi}$ to a new clock assignment where*

$$\vec{\pi}[\eta := 0](\vec{\pi})_i \triangleq \begin{cases} 0 & (\xi_i \in \eta) \\ \vec{\pi}_i & (\xi_i \notin \eta) \end{cases}$$

- **region**: *A region ρ is a connected subset of $\mathbb{R}^n$ formed by a conjunction of clock constraints. Let $\mathcal{R}$ be the set of regions in $\mathbb{R}^n$.*

- **input action—name**: $a \in \mathcal{A}$.

- **output action—coname**: $\overline{a} \in \overline{\mathcal{A}}, \overline{\overline{a}} = a$, *(also, $\overline{a} = a\_$ in this work).*

- **Labels**: $\mathbb{L} = \mathcal{A} \cup \overline{\mathcal{A}}$.

- $\tau \notin \mathbb{L}$ *the invisible internal action.*

- **location**: $\langle l, \rho_l \rangle$, *where l is unique location name, and ρ_l is a past-closed region called a* **location invariant**. *A region ρ is* **past-closed** *when it includes time $\vec{0}$ i.e., given that $\vec{p} \leqslant \vec{q} \Leftrightarrow \forall\, i \in [1..n][\vec{p}_i \leq \vec{q}_i]$ then*

$$\forall\, \vec{p} \in \rho[\forall\, \vec{d} \in \mathbb{R}^n[\vec{0} \leqslant \vec{d} \leqslant \vec{p} \Rightarrow \vec{d} \in \rho]]$$

Note that only clock constraints with $R \in \{\leq, <\}$ result in past-closed regions.

A TSA $\mathcal{T} \in \mathbb{T}$ is a 5-tuple $\mathcal{T} = \langle \mathcal{L}, Act, \Xi, \langle l_0, \rho_0 \rangle, \longmapsto \rangle$, where

- *$\mathcal{L}$ is a finite set of locations.*

- *$Act = \mathbb{L} \cup \{\tau\}$ is a set of actions ranged over by α.*

- *$\Xi \subseteq C$ is a set of n $\mathbb{R}$-valued clocks.*

- *$\langle l_0, \rho_0 \rangle \in \mathcal{L}$ is the start location, where initially $\vec{\pi} \in \rho_0 = \vec{0}$.*

- $\longmapsto \subseteq \mathcal{L} \times Act \times \mathcal{R} \times \wp(\Xi) \times \mathcal{L}$ *is a transition relation, where each transition is labeled by an action, a region (called a* **guard***), and a set of clocks that are reset to* 0 *when the transition occurs (note* $\wp(\Xi)$ *denotes the powerset of* Ξ*).*

Transition guards are derived from clock constraints, and they are interpreted as necessary conditions for the transition to occur. Location invariants are also derived from clock constraints, and they restrict the amount of time the automata can stay in the associated location. Location invariants are therefore interpreted as sufficient conditions to cause a transition to occur (HNSY94:209). They cause transitions to occur when time passing forces a change of location to avoid $\vec{\pi} \notin \rho_l$. Location invariants are also necessary conditions for the TSA to be in the associated location. Unspecified (empty) guards and invariants are defined to be the region $\mathbb{R}^n$ (always satisfied). Informally, TSA operate by taking instantaneous transitions from location to location. When no transitions occur, TSA idle in a location $\langle l, \rho_l \rangle$ passing time by incrementing all clocks $x_i \in \vec{\pi}$ by $d \in \mathbb{R}$ such that l's location invariant is satisfied—i.e., $\vec{\pi} + d \in \rho_l$. Without loss of generality, only non-Zeno TSA are considered. Non-Zenoness is a liveness condition that asserts time can always progress (HNSY94:203). No generality is lost by excluding Zeno automata because any well-formed Zeno TSA can be transformed to a non-Zeno one by strengthening invariants. Non-Zeno automata consistently model the fact that time relentlessly progresses.

3.2 TSA Semantics

The semantics of TSA are defined via Dense Labeled Transition System (DLTS) automata with uncountable state sets.

Definition 15. TSA Semantics. *Let* $\mathbb{D}$ *denote the set of DLTS automata. Every TSA* $\mathcal{T} = \langle \mathcal{L}, Act, \Xi, \langle l_0, \rho_0 \rangle, \longmapsto \rangle$ *induces a DLTS automaton* $\mathcal{D} = \langle S, Act, \longrightarrow, \langle l_0, \vec{0} \rangle \rangle$ *such that:*

- *S is a set of timed states defined by the following rule:*

$$\forall \langle l, \rho_l \rangle \in \mathcal{L}[\vec{\pi} \in \rho_l \Rightarrow \langle l, \vec{\pi} \rangle \in S] \tag{5}$$

  $S_I \subseteq S$ and $S_S \subseteq S$ are sometimes used to distinguish between the implementation and specification DLTS state spaces.

- *$Act = \mathbb{L} \cup \{\tau\}$ a set of actions ranged over by α.*

- *$\langle l_0, \vec{0} \rangle$ the start state assigning 0 to every clock.*

- *$\longrightarrow \subseteq S \times (Act \cup \ \mathbb{R}) \times S$ is a transition relation defined by the following two rules:*

$$\langle l, \rho_l \rangle \overset{\alpha,\rho,\eta}{\longmapsto} \langle l', \rho_{l'} \rangle \wedge \vec{\pi} \in \rho \wedge \vec{\pi} \in \rho_l \wedge \vec{\pi}[\eta := 0] \in \rho_{l'} \quad \Rightarrow \quad \langle l, \vec{\pi} \rangle \xrightarrow{\alpha} \langle l', \vec{\pi}[\eta := 0] \rangle \tag{6}$$

$$\langle l, \rho_l \rangle \in S \wedge \delta \in \mathbb{R} \wedge \vec{\pi}, \vec{\pi} + \delta \in \rho_l \quad \Rightarrow \quad \langle l, \vec{\pi} \rangle \xrightarrow{\delta} \langle l, \vec{\pi} + \delta \rangle \tag{7}$$

In Rule 6, DLTS $\mathcal{D}$ transitions from location l to l' via action α. No time passes, but all clocks in $\eta \subseteq \Xi$ are reset to 0. Clock assignment $\vec{\pi}$ must satisfy both ρ_l and ρ, and clock reset $\vec{\pi}[\eta := 0]$ must satisfy $\rho_{l'}$. Under rule 6, timed state $\langle l', \vec{\pi}[\eta := 0] \rangle$ is a **transition successor** of timed state $\langle l, \vec{\pi} \rangle$.

In Rule 7, DLTS $\mathcal{D}$ stays in location l with time delay δ if both $\vec{\pi}$ and $\vec{\pi} + \delta$ satisfy ρ_l. Under rule 7, timed state $\langle l', \vec{\pi} + \delta \rangle$ is a **time successor** of timed state $\langle l, \vec{\pi} \rangle$.

3.3 TSA Modifications

As in process algebras, TSA transition relations can be modified to generate new TSA. The named process-algebra-style rules for generating new TSA are defined as follows:

Definition 16. TSA Modification Rules. *Let $A \in \mathcal{L}, L \subseteq \mathbb{L}$, then*

$$\forall\ \alpha \in \mathbb{L}, \bar{l} \in \overline{\mathcal{A}}$$

$$\textit{Res:}\quad \frac{A \overset{\alpha,\rho,\eta}{\longmapsto} A'}{A \setminus L \overset{\alpha,\rho,\eta}{\longmapsto} A' \setminus L}(\alpha \notin L)$$

$$\textit{Hid:}\quad \frac{A \overset{\bar{l},\rho,\eta}{\longmapsto} A'}{A/L \overset{\tau,\rho,\eta}{\longmapsto} A'/L}(\bar{l} \in L \cup \overline{L})$$

$$\textit{Rel:}\quad \frac{A \overset{\alpha,\rho,\eta}{\longmapsto} A'}{A[f] \overset{f(\alpha),\rho,\eta}{\longmapsto} A'[f]}$$

Rule **Res** deletes transitions by restricting the transition relation by a set of actions $L \subseteq \mathbb{L}$ (recall that $\mathbb{L}$ is the set of visible actions not including τ or $\delta \in \mathbb{R}$). Only actions not in L remain in $(A \setminus L)$'s transition relation.

Rule **Hid** turns output actions into hidden actions (τ's). This rule makes it impossible for other TSA to interface with A/L using l. Hid does not delete any transitions from A; it just relabels outputs whose name or coname are members of L. Hid is particularly important for internalizing the cooperative actions of parallel composed TSA as discussed in Section 3.4.

Rule **Rel** relabels transitions. The change is specified by a function $f : \mathbb{L} \to \mathbb{L}$, where the convention is to specify f by a list of label pairs $\langle \text{new}, \text{old} \rangle$ relating old labels of A with the new labels of $A[f]$. Relabeling does not change inputs into outputs or vice-versa; i.e., for $n, o \in \mathbb{L}$ and $\langle n, o \rangle$, a relabeling specification pair, then $f(o) = n$, and $f(\bar{o}) = \bar{n}$.

3.4 Parallel TSA Composition

Specifying the behavior of complex implementations as single flat TSA is too tedious. Generally, in hardware and software design processes, systems are built and understood hierarchically; i.e., the entire system is composed of subsystems, which are composed of sub-subsystems, which are composed of sub-sub-subsystems, ..., until at the lowest level of abstraction simple well defined design primitives are used. Modern structured analysis leads to hierarchical software systems, and the design primitives are typically programming language statements or library functions and procedures. The process is similar for hardware systems, except that the primitives are logic gates, transistors, or standard cells. Logically, designers usually think of the components functioning in parallel, generally independent of each other except for the specific dependencies implied by their connections with each other. However, if the system is a software system on a uniprocessor, the actual processes may run serially or in a time-sharing environment.

Figures 4 and 5 show the schematic for a C-element implementation and a corresponding Timed Logic Conformance System (TLCS) parallel TSA specification.

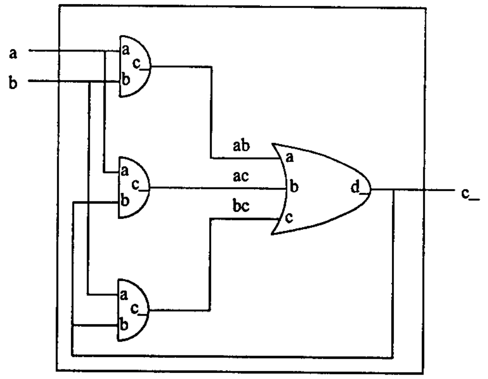


Figure 4. C-element Schematic.

```
tsa([c_elt000,AndMin,AndMax,OrMin,OrMax],CE) :-
   parallel([[[and000,AndMin,AndMax],[[ab,c]]],
             [[and000,AndMin,AndMax],[[c,b], [ac,c]]],
             [[and000,AndMin,AndMax],[[b,a], [c,b], [bc,c]]],
             [[or0000,OrMin,OrMax], [[ab,a], [ac,b], [bc,c], [c,d]]]],
            [ab, ac, bc],
            CE).
```

Figure 5. Parallel C-element Example.

The C-element is composed of three 2-input *Ands*, and a single 3-input *Or*. The timing of the *And* and *Or* TSA are specified with the variables `AndMin`, `AndMax` and `OrMin`, `OrMax`. The list of pairs following each component instantiation (e.g., `[new,old]`) rename the default input and output names to the new names in the circuit. The list of names following the list of components (i.e., `[ab, ac, bc]`) are the hidden internal connections of the C-element; they are not available for connection to the world outside of C-element, and they become the internal τ actions of the C-element component.

Formally, the relationship between the parallel TSA definition and the definitions of its components is as follows:

Definition 17. Parallel TSA. *Let $\mathcal{T}_p = \langle \mathcal{L}_p, Act_p, \Xi_p, \langle l_0, \rho_0 \rangle_p, \longmapsto_p \rangle$ be the parallel TSA constructed of n TSA, each distinctly denoted by $\mathcal{T}_i = \langle \mathcal{L}_i, Act_i, \Xi_i, \langle l_0, \rho_0 \rangle_i, \longmapsto_i \rangle$ then,*

$$\mathcal{L}_p \subseteq \mathcal{L}_1 \times \mathcal{L}_2 \times \ldots \times \mathcal{L}_n \tag{8}$$

$$Act_p \subseteq \bigcup_{i \in \{1 \ldots n\}} Act_i \tag{9}$$

$$\Xi_p \subseteq \bigcup_{i \in \{1 \ldots n\}} \Xi_i \tag{10}$$

$$\langle l_0, \rho_0 \rangle_p \triangleq \langle \langle l_0, \rho_0 \rangle_1, \langle l_0, \rho_0 \rangle_2, \ldots, \langle l_0, \rho_0 \rangle_n \rangle \tag{11}$$

$$\longmapsto_p \subseteq \mathcal{L}_p \times Act_p \times \mathcal{R} \times \wp(\Xi_p) \times \mathcal{L}_p \tag{12}$$

The locations of the parallel TSA (elements of the set $\mathcal{L}_p$) are denoted by a sequence of locations of its subcomponents called a **location vector**. The initial location vector of the parallel TSA ($\langle l_0, \rho_0 \rangle_p$) consists of the initial locations of every component. Except for the initial location vector, $\mathcal{T}_p$'s definition is not complete; the exact subsets are defined inductively based on the initial location, subcomponent definitions, and the Definition 18 named rules governing communication between $\mathcal{T}_p$'s subcomponents. Parallel TSA location vector invariants are logically the intersection of time regions formed by conjuncting the clock constraints of the sub-component location invariants. The transition relation $\longmapsto_p$ is derived from the transition relations of the subcomponents using the rules starting from the initial location vector $\langle l_0, \rho_0 \rangle_p$. The set of actions of the parallel composition Act_p are also derived from $\cup Act_i$ as defined by induction over the rules. Reached location-vectors are added to set $\mathcal{L}_p$, and new actions possible from added locations are added to set Act_p. Clocks referenced in added locations and new transitions are all included in set Ξ_p.

Definition 18. Parallel TSA Composition Rules. *Let $i, j \in [1..n], i \neq j, A \in S_i, B \in S_j$ be locations of two subcomponents in the parallel TSA $\mathcal{T}_p$, then the sets of locations $\mathcal{L}_p$, actions Act_p, and transitions $\longmapsto_p$ are defined inductively by rules:*

$$\forall\, l \in \mathcal{A}, \bar{l} \in \overline{\mathcal{A}}, \sigma \in Act$$

$$\textit{Single1:} \quad \frac{A \overset{\sigma,\rho,\eta}{\longmapsto}_i A'}{A \| B \overset{\sigma,\rho,\eta}{\longmapsto}_p A' \| B} (\sigma, \overline{\sigma} \notin \mathbb{L}_B)$$

$$\textit{Single2:} \quad \frac{B \overset{\sigma,\rho,\eta}{\longmapsto}_j B'}{A \| B \overset{\sigma,\rho,\eta}{\longmapsto}_p A \| B'} (\sigma, \overline{\sigma} \notin \mathbb{L}_A)$$

$$\textit{Com1:} \quad \frac{A \overset{l,\rho_a,\eta_a}{\longmapsto}_i A' \wedge B \overset{l,\rho_b,\eta_b}{\longmapsto}_j B'}{A \| B \overset{l,\rho_a \cap \rho_b,\eta_a \cup \eta_b}{\longmapsto}_p A' \| B'}$$

$$\textit{Com2:} \quad \frac{A \overset{\bar{l},\rho_a,\eta_a}{\longmapsto}_i A' \wedge B \overset{l,\rho_b,\eta_b}{\longmapsto}_j B'}{A \| B \overset{\bar{l},\rho_a \cap \rho_b,\eta_a \cup \eta_b}{\longmapsto}_p A' \| B'}$$

$$\textit{Com3:} \quad \frac{A \overset{l,\rho_a,\eta_a}{\longmapsto}_i A' \wedge B \overset{\bar{l},\rho_b,\eta_b}{\longmapsto}_j B'}{A \| B \overset{\bar{l},\rho_a \cap \rho_b,\eta_a \cup \eta_b}{\longmapsto}_p A' \| B'}$$

Output alphabets of the component TSA must be disjoint; i.e.,

$$\forall\, i, j \in [1, n][i \neq j \Rightarrow ((\overline{\mathcal{A}}_i \cap \overline{\mathcal{A}}_j) = \varnothing)]$$

More than one component outputting the same action is considered an error and the composition is not defined.

Although the parallel composition rules are specified for only two TSA locations at a time, they are commutative and associative[1] and extend by composing two TSA locations at a time to produce a new parallel TSA location which is again composed with the adjacent TSA locations until the composition is complete.

[1] The Boolean and set operations are all commutative and associative, and the rules are specified in symmetric pairs where necessary.

TSA locations A and B composed by rules **Single1** and **Single2** continue to perform internal actions and visible actions not in each other's language by name or coname. None of the actions for the gate-level TSA composed together in Figure 4 are independent of each other, so rules Single1 and Single2 do not induce any C-element transitions in this example.

Rule **Com1** allows multiple TSA to input together. Whenever the parallel TSA is in a from-location where the locations of all components sharing a common input action α can perform an α action, there is a parallel TSA α action to a to-location vector where all components sharing the α action are updated to the to-locations of their α transitions, and the non-α-capable component locations are equal in the from and to-location vectors. Receiver transition guards are conjuncted (i.e., their regions ρ_a and ρ_b are intersected), and reset sets are unioned together in the parallel action. This case is illustrated by the a-input shared by the top two *Ands* in Figure 4. Whenever both *Ands* (all receivers) are in a location that can perform an a, there is a parallel TSA a action to the new location vector where both *Ands* (all a receivers) move to their a-transition destination locations. If one or more receivers cannot perform the shared input action, no parallel transition is defined.

Rules **Com2** and **Com3** generate output actions when two or more TSA cooperate on an output and its complementary input action. Used with the Hid TSA modification rule, Com2 and Com3 internalize cooperative actions, turning them into τ's. When the parallel output action is not hidden, it remains an output of the composition. The unhidden case is illustrated by the *Or* $\bar{c}$ output and the c-input shared by the bottom two *Ands* in Figure 4. Whenever all three gates are in a location where they can perform the $\bar{c}$ or c action, there is a parallel TSA $\bar{c}$ action where all three gates move to their $\bar{c}$-c-destination locations in the location vector. By convention, **Com2** and **Com3** keep the coname label; this supports building parallel TSA that can export an output that communicates to TSA in the parallel composition as well as those external to the composition. The hidden action case is illustrated by the top *And* $\overline{ab}$ output and the *Or* ab-input in Figure 4.

Whenever the top *And* is in a location where it can perform the $\overline{ab}$ and the *Or* can perform the ab action, there is a parallel TSA $\tau(ab)$ action where both gates move to their $\overline{ab}$-ab-destination locations in the location vector.

This formalization of parallel composition is synchronous since it does not allow the individual σ-actions of one component to occur independently of other components when they can also perform σ or $\overline{\sigma}$. This means that they strongly influence each other's behavior, but it faithfully models the reality of hardware components connected by wires strongly influencing each other.

3.5 Summary

This chapter formally defines a "simple" and expressive Timed Safety Automata (TSA) model of computation. It includes basic TSA definitions, TSA semantics, rules for modifying TSA, and rules and examples defining parallel TSA composition. TSA are well suited to modeling hardware components because they does not suffer the deficiencies recognized in Chapter II.

This Mealy machine TSA model is simpler than the Moore machine COSPAN timed process model because it does not define output by associating functions with locations, and it requires about half of the rules CTR requires (10 vs. 21) to define model semantics and composition.

TSA suffer none of the expressiveness problems associated with untimed process algebras, TCCS and CTR. Upper and lower time bounds (bi-bounded delays) are easily defined using TSA location invariants and transition guards. The maximal-progress semantic leap (from two processes waiting individually to perform their actions to cooperating processes that can not wait to perform their cooperative actions) does not exist in the Definition 18 TSA parallel composition rules. And general temporal relationships between actions that do not sequentially follow each other are easy to express in TSA by resetting a clock and freely using clock predicates to define the relationship.

IV. Timed Logic Conformance

Following Ken Stevens' bisimulation-based Logic Conformance relation $\succeq_l$ (Ste94), this chapter defines a timed relation called Timed Logic Conformance (TLC, also written as ${}_{o}\underline{\prec\!\succ}_{i}$) for Timed Safety Automata (TSA) based on DLTS semantics. TLC enforces a time-interval-based relationship between times when implementation actions can occur relative to specification actions. It also maintains $\succeq_l$'s partial order relationship between specification and implementation actions. TLC loosens the standard bisimulation-based strict timed-equivalence requirement formalized by Wang (Wan90), Čerāns (Cer92), Alur, Courcoubetis, Henzinger (ACH94), and others (LY93). Instead of strict timed-equivalence a partial order relating the states of two systems over the time intervals when actions are enabled is defined. The partial order requires that implementation inputs are a timed superset of specification inputs[1], and that implementation outputs are a timed subset of specification outputs. For example, ${}_{o}\underline{\prec\!\succ}_{i}$ will relate TSA implementation (I) and specification (S) such that $I \; {}_{o}\underline{\prec\!\succ}_{i} \; S$ iff $I \succeq_l S$ and all output actions of I occur within the time intervals observed for S's output actions and all input actions of S occur within the time intervals observed for I's input actions.

TLC is different from other loose timed-refinement relations (ACD90, Dan92, CGL93, Cer95). In particular, ${}_{o}\underline{\prec\!\succ}_{i}$ turns around the standard definition that typically requires implementation input actions to be a timed subset of specification input actions. This change is motivated by common sense that argues one cannot safely substitute an implementation that does not accept all of the inputs accepted by the specification. TLC does not require designers to specify behaviors for all possible inputs in all locations at all times and it allows implementations that accept more inputs than the specification. In contrast to the assumes-guarantees verification methodology, TLC supports declaring the input constraints of the specification and implementations and using them to decompose the problem into independent pieces in a simple and powerful way. It does not

[1] Exceptions to the $I \succeq_i S$ half of the TLC relationship are allowed under certain circumstances; see Def. 28.

require many different abstract models of each component's environment or iterating over extra verifications to "verify the verification."

Before defining the TLC relation itself, the next section leads up to it by defining how to abstract internal structural differences between TSA. Section 4.2 defines a weak timed bisimulation equivalence relation that will be used later to show that TLC is a partial order. After that, Section 4.3 defines how to abstract temporal differences between TSA. Then Section 4.4 defines TLC, and Section 4.5 explains an example. Section 4.6 compares TLC to other relations. Sections 4.7 and 4.8 define and prove the necessary properties of the TLC relation. Finally, Section 4.9 discusses the TLC verification methodology.

4.1 Abstracting Internal Differences

As for $\succeq_I$, internal behavior is abstracted into τ-transitions, and internal state changes that are matched by a TSA staying in an equivalent state are ignored. Recall $\tau \in Act$ is a distinguished element of *Act*, and let hatted Greek letters like $\widehat{\alpha}$ formalize when τ actions may sometimes be matched by staying in the same state and passing zero time as follows:

Definition 19. τ-abstraction: $\widehat{\alpha}$.

$$\forall_{\alpha \in Act \cup \mathbb{R}}\ \widehat{\alpha} \triangleq \begin{cases} 0, & \mathit{if}\ \alpha = \tau \\ \alpha, & \mathit{if}\ \alpha \neq \tau \end{cases}$$

To further loosen implementation and specification action-matching requirements, the transition relations of the systems are extended by transitively closing them over certain action sequences.

Definition 20. τ-closure: *$P \overset{\sigma}{\Longrightarrow} Q$. A DLTS transition relation $R \subseteq (S \times (Act \cup \mathbb{R}) \times S)$ is τ-transitive if whenever*

$$P(\xrightarrow{\tau})^* \xrightarrow{\sigma} (\xrightarrow{\tau})^* Q \quad \wedge \quad \sigma \in Act \cup \mathbb{R}\ \vee$$

$$P \xrightarrow{\delta_1} (\xrightarrow{\tau})^* \xrightarrow{\delta_2} Q \quad \wedge \quad \sigma = \delta_1 + \delta_2$$

exists in R then $P \xrightarrow{\sigma} Q$ also exists in R.

The τ-closure of a DLTS transition relation $R \subseteq (S \times (Act \cup \mathbb{R}) \times S)$, is the relation R' such that

1. *R' is τ-transitive.*
2. *$R' \supseteq R$.*
3. *For any τ-transitive relation R'', $R'' \supseteq R \Rightarrow R'' \supseteq R'$.*

Transitions from state P to state Q by action σ in τ-closure are denoted by $P \overset{\sigma}{\Longrightarrow} Q$. The predicate $P \overset{\sigma}{\Longrightarrow}$ is true when there is at least one transition from state P via action $\sigma \in Act \cup \mathbb{R}$. No actions are time abstracted in τ-closure, but the τ-closure relation models tau-abstracted actions. The τ-closure is used to extend transition relations and ignore internal actions resulting from structural differences that do not matter.

4.2 Weak Timed Bisimulation

Weak Timed Bisimulation is an equivalence relation for DLTS automata that will shortly be used to show that Timed Logic Conformance is a partial order.

Definition 21. Weak Timed Bisimulation: $\mathcal{W}$.

A binary relation $\mathcal{W} \subseteq S_I \times S_S$ over DLTS automata states is a weak timed bisimulation between two DLTS's $\langle S_I, Act_I, \longrightarrow_I, \langle l_0, \pi_0 \rangle_I \rangle$, and $\langle S_S, Act_S, \longrightarrow_S, \langle l_0, \pi_0 \rangle_S \rangle$, iff

$$\forall \langle I, S \rangle \in \mathcal{W}, \gamma \in Act \cup \mathbb{R} \; [$$

$$\forall S' \, [S \xrightarrow{\gamma} S' \;\Rightarrow\; \exists I' \, [I \overset{\hat{\gamma}}{\Longrightarrow} I' \wedge \langle I', S' \rangle \in \mathcal{W}]] \wedge \tag{13}$$

$$\forall I' \, [I \xrightarrow{\gamma} I' \;\Rightarrow\; \exists S' \, [S \overset{\hat{\gamma}}{\Longrightarrow} S' \wedge \langle I', S' \rangle \in \mathcal{W}]]] \tag{14}$$

Some properties of weak timed bisimulation are preserved by various operations on relations over DLTS state spaces. Let the identity Id, converse $\mathcal{R}^{-1}$ of a binary relation $\mathcal{R}$, and the composition

$\mathcal{R}_1\mathcal{R}_2$ of binary relations be defined as follows:

$$Id \triangleq \{\langle x,x\rangle \mid x \in S\} \tag{15}$$

$$\mathcal{R}^{-1} \triangleq \{\langle x,y\rangle \mid \langle y,x\rangle \in \mathcal{R}\} \tag{16}$$

$$\mathcal{R}_1\mathcal{R}_2 \triangleq \{\langle p,r\rangle \mid \langle p,q\rangle \in \mathcal{R}_1 \wedge \langle q,r\rangle \in \mathcal{R}_2\} \tag{17}$$

Lemma 1 *Assume that each $\mathcal{W}_i$ $(i = 1, 2, \ldots)$ is a weak timed bisimulation, then the following relations are all weak timed bisimulations.*

(1) Id

(2) $\mathcal{W}_i^{-1}$

(3) $\mathcal{W}_1\mathcal{W}_2$

(4) $\bigcup \mathcal{W}_i$

Proof

1. *Id*: $\forall\ P \in S, \gamma \in Act \cup \mathbb{R}$ each transition $P \xrightarrow{\gamma} P'$ can be matched by itself in the superset transition relation $P \overset{\gamma}{\Longrightarrow} P'$, and $P \xrightarrow{\tau} P' \Rightarrow (P \xrightarrow{0} P \xrightarrow{\tau} P' \xrightarrow{0} P') \Rightarrow (P \overset{0}{\Longrightarrow} P \overset{\tau}{\Longrightarrow} P' \overset{0}{\Longrightarrow} P') \Rightarrow (P \overset{0}{\Longrightarrow} P') \Rightarrow P \overset{\hat{\tau}}{\Longrightarrow} P'$, therefore, $(\langle P,P\rangle \in Id \wedge P \xrightarrow{\gamma} P') \Rightarrow P \overset{\hat{\gamma}}{\Longrightarrow} P' \langle P',P'\rangle \in Id$, and therefore, Id is a weak timed bisimulation.

2. $\mathcal{W}_i^{-1}$: Given any weak timed bisimulation $\mathcal{W}_i$ $\forall\ \langle S,I\rangle \in \mathcal{W}_i^{-1}, \gamma \in Act \cup \mathbb{R}$ all transitions $S \xrightarrow{\gamma} S'$ and $I \xrightarrow{\gamma} I'$ are matched by transitions $I \overset{\hat{\gamma}}{\Longrightarrow} I'$ and $S \overset{\hat{\gamma}}{\Longrightarrow} S'$ and $\langle S',I'\rangle \in \mathcal{W}_i^{-1}$ therefore $\mathcal{W}_i^{-1}$ is a weak timed bisimulation.

3. $\mathcal{W}_1\mathcal{W}_2$: Given two weak timed bisimulations, $\mathcal{W}_1$ and $\mathcal{W}_2$, and the composition of those bisimulations, $\mathcal{W}_1\mathcal{W}_2$, the proof proceeds by assuming $\langle P,Q\rangle \in \mathcal{W}_1 \wedge \langle Q,R\rangle \in \mathcal{W}_2 \wedge \langle P,R\rangle \in \mathcal{W}_1\mathcal{W}_2$ and showing that for all possible actions that must be matched in Formulas 13 and 14 in Def. 21, the actions are matched across the composition and $\langle P',R'\rangle \in \mathcal{W}_1\mathcal{W}_2$.

(a) Formula 14: Since $\mathcal{W}_1$ is a weak timed bisimulation all transitions $P \xrightarrow{\gamma} P'$ are matched by transitions $Q \xRightarrow{\widehat{\gamma}} Q'$.

   i. $Q(\xrightarrow{\tau})^*Q^m \xrightarrow{\widehat{\gamma}} Q^n(\xrightarrow{\tau})^*Q'$ (Def. 20, first disjunct): Since $\mathcal{W}_2$ is a weak timed bisimulation , according to Formula 14, every Q^1, $Q \xrightarrow{\tau} Q^1$ is matched by some R^1 such that $R \xRightarrow{\widehat{\tau}} R^1 \wedge \langle Q^1, R^1 \rangle \in \mathcal{W}_2$. This is true inductively for more $\tau \in \tau^*$ and $Q^1 \cdots Q^m$ and $R^1 \cdots R^m$ according to Formula 14. According to Formula 14 whether $\widehat{\gamma} = 0$ or $\widehat{\gamma} = \gamma$, $Q^m \xrightarrow{\widehat{\gamma}} Q^n$ is matched by some R^n such that $R^m \xRightarrow{\widehat{\gamma}} R^n \wedge \langle Q^n, R^n \rangle \in \mathcal{W}_2$. Finally, by Formula 14 any sequence of τ's in $Q^n \xrightarrow{\tau^*} Q'$ is matched by $R^n \xRightarrow{\widehat{\tau^*}} R' \wedge \langle Q', R' \rangle \in \mathcal{W}_2$. And, by Def. 20, $(R \xRightarrow{\widehat{\tau^*}} R^m \xRightarrow{\widehat{\gamma}} R^n \xRightarrow{\widehat{\tau^*}} R') \Rightarrow R \xRightarrow{\widehat{\gamma}} R'$

   ii. $Q \xrightarrow{\delta_1} Q^m(\xrightarrow{\tau})^*Q^n \xrightarrow{\delta_2} Q' \wedge \gamma = \delta_1 + \delta_2$ (Def. 20, second disjunct): Since $\mathcal{W}_2$ is a timed logic conformation, according to Formula 14, Q^m, $Q \xrightarrow{\delta_1} Q^m$ is matched by some R^m such that $R \xRightarrow{\widehat{\delta_1}} R^m \wedge \langle Q^m, R^m \rangle \in \mathcal{W}_2$. Likewise, by Formula 14 any sequence of τ's in $Q^m \xrightarrow{\tau^*} Q^n$ is matched by $R^m \xRightarrow{\widehat{\tau^*}} R^n \wedge \langle Q^n, R^n \rangle \in \mathcal{W}_2$. Finally, Formula 14 ensures that all Q', $Q^n \xrightarrow{\delta_2} Q'$ are matched by some R' such that $R^n \xRightarrow{\widehat{\delta_2}} R' \wedge \langle Q', R' \rangle \in \mathcal{W}_2$. And, by Def. 20, $(R \xRightarrow{\delta_1} R^m \xRightarrow{\widehat{\tau^*}} R^n \xRightarrow{\delta_2} R') \Rightarrow R \xRightarrow{\widehat{\gamma}} R'$

   Therefore, $R \xRightarrow{\widehat{\gamma}} R' \wedge \langle P', R' \rangle \in \mathcal{W}_1\mathcal{W}_2$ by the definition of composition (Formula 17).

(b) Formula 13: By reasoning from right to left in the same way, all transitions $R \xrightarrow{\gamma} R'$ are matched by transitions $Q \xRightarrow{\widehat{\gamma}} Q'$ and $P \xRightarrow{\widehat{\gamma}} P' \wedge \langle P', R' \rangle \in \mathcal{W}_1\mathcal{W}_2$, so $\mathcal{W}_1\mathcal{W}_2$ is a weak timed bisimulation.

4. $\bigcup \mathcal{W}_i$: $\forall \langle I, S \rangle \in \bigcup \mathcal{W}_i, \gamma \in Act \cup \mathbb{R}$ all transitions $S \xrightarrow{\gamma} S'$ and $I \xrightarrow{\gamma} I'$ are matched by transitions $I \xRightarrow{\widehat{\gamma}} I'$ and $S \xRightarrow{\widehat{\gamma}} S'$ from some $\mathcal{W}_i$ in the union and $\langle I', S' \rangle \in \mathcal{W}_i \Rightarrow \langle I', S' \rangle \in \bigcup \mathcal{W}_i$ therefore $\bigcup \mathcal{W}_i$ is a weak timed bisimulation. □

Def. 21 does not uniquely identify a particular relation (e.g., $\varnothing$ is a weak timed bisimulation). The definition is strengthened here by referring to the largest weak timed bisimulation (i.e., maximal fixpoint) or union of timed bisimulations.

Definition 22. Weak Timed Bisimulation Maximum Fixpoint: $\widehat{\mathcal{W}}$.

Given two DLTS's $\langle S_I, Act_I, \longrightarrow_I, \langle l_0, \pi_0 \rangle_I \rangle$, and $\langle S_S, Act_S, \longrightarrow_S, \langle l_0, \pi_0 \rangle_S \rangle$,

$$\widehat{\mathcal{W}} \triangleq \bigcup_{\mathcal{R} \in \wp(S_I \times S_S)} \mathcal{R} \textit{ is a weak timed bisimulation}$$

Theorem 1 *Given Def. 22, $\widehat{\mathcal{W}}$ is the largest weak timed bisimulation.*

Proof

By Lemma 1(4), $\widehat{\mathcal{W}}$ is a weak timed bisimulation and by definition it includes any other such.

□

Now DLTS automata are related to one another using maximum weak timed bisimulations.

Definition 23. Weak Timed Bisimilar DLTS: $\approx$.

Two DLTS's $I \triangleq \langle S_I, Act_I, \longrightarrow_I, \langle l_0, \pi_0 \rangle_I \rangle$, and $S \triangleq \langle S_S, Act_S, \longrightarrow_S, \langle l_0, \pi_0 \rangle_S \rangle$, are weak timed bisimilar (written $I \approx S$) iff

$$\langle \langle l_0, \pi_0 \rangle_I, \langle l_0, \pi_0 \rangle_S \rangle \in \widehat{\mathcal{W}}$$

Theorem 2 *Given Def. 22, $\approx$ is an equivalence relation.*

Proof

1. Reflexivity: For any DLTS P, $P \approx P$ by Lemma 1(1) since $\langle \langle l_0, \pi_0 \rangle_P, \langle l_0, \pi_0 \rangle_P \rangle \in \mathcal{W}$.

2. Symmetry: For two DLTS P and Q, $P \approx Q \Rightarrow Q \approx P$ since $\langle \langle l_0, \pi_0 \rangle_P, \langle l_0, \pi_0 \rangle_Q \rangle \in \widehat{\mathcal{W}} \Rightarrow \langle \langle l_0, \pi_0 \rangle_Q, \langle l_0, \pi_0 \rangle_P \rangle \in \widehat{\mathcal{W}}^{-1}$ per Lemma 1(2) .

3. Transitivity: For three DLTS P, Q, and R, $P \approx Q \wedge Q \approx R \Rightarrow P \approx R$ by Lemma 1(3) since $\langle \langle l_0, \pi_0 \rangle_P, \langle l_0, \pi_0 \rangle_Q \rangle \in \widehat{\mathcal{W}_1} \wedge \langle \langle l_0, \pi_0 \rangle_Q, \langle l_0, \pi_0 \rangle_R \rangle \in \widehat{\mathcal{W}_2} \Rightarrow \langle \langle l_0, \pi_0 \rangle_P, \langle l_0, \pi_0 \rangle_R \rangle \in \widehat{\mathcal{W}_1 \mathcal{W}_2}$. □

Now, the equivalence relation $\approx$ between DLTS is established. It relates two DLTS that are observationally equivalent with respect to their external actions despite the fact that they may have significantly different internal action sequences. Weak timed bisimulation does not yet allow the timing of two DLTS to vary.

4.3 Abstracting Temporal Differences

In order to allow the inputs of the implementation to be a timed superset of specification inputs, and the outputs of the specification to be a timed superset of implementation outputs, further abstraction operations are defined by closing transition relations over time-passing actions under certain conditions.

Definition 24. Input-δ-τ-Closure: $P \overset{\sigma}{\Longrightarrow}_i Q$. *A DLTS transition relation $R \subseteq (S \times (Act \cup \mathbb{R}) \times S)$ is input-δ-τ-transitive if whenever*

$$\begin{aligned} P(\xrightarrow{\tau})^* \xrightarrow{\sigma} (\xrightarrow{\tau})^* Q \quad &\wedge \quad \sigma \in Act \cup \mathbb{R} \ \vee \\ P \xrightarrow{\delta_1} (\xrightarrow{\tau})^* \xrightarrow{\delta_2} Q \quad &\wedge \quad \sigma = \delta_1 + \delta_2 \ \vee \\ P \xrightarrow{\delta_1} \xrightarrow{\sigma} \xrightarrow{\delta_2} Q \quad &\wedge \quad \sigma \in \mathcal{A}, \delta_1, \delta_2 \in \mathbb{R} \end{aligned}$$

exists in R then $P \xrightarrow{\sigma} Q$ also exists in R.

The input-δ-τ-closure of a DLTS transition relation $R \subseteq (S \times (Act \cup \mathbb{R}) \times S)$, is the relation R' such that

1. *R' is input-δ-τ-transitive.*
2. *$R' \supseteq R$.*
3. *For any input-δ-τ-transitive relation R'', $R'' \supseteq R \Rightarrow R'' \supseteq R'$.*

Transitions from state P to state Q by action σ in input-δ-τ-closure are denoted by $P \overset{\sigma}{\Longrightarrow}_i Q$. The predicate $P \overset{\sigma}{\Longrightarrow}_i$ is true when there is at least one transition from state P via action $\sigma \in Act \cup \mathbb{R}$. Input-$\delta$-$\tau$-closure models time-and-tau-abstracted input actions. Outputs, τ, and δ actions themselves are not time abstracted. Input-δ-τ-closure extends specification transition relations to match implementation behaviors, but it does not allow the timing of outputs, δ's, or τ's to vary.

Definition 25. Output-δ-τ-Closure: $P \overset{\sigma}{\Longrightarrow}_o Q$. *A DLTS transition relation* $R \subseteq (S \times (Act \cup \mathbb{R}) \times S)$ *is output-δ-τ-transitive if whenever*

$$\begin{aligned} P(\xrightarrow{\tau})^* \xrightarrow{\sigma} (\xrightarrow{\tau})^* Q &\quad \wedge \quad \sigma \in Act \cup \mathbb{R} \ \vee \\ P \xrightarrow{\delta_1} (\xrightarrow{\tau})^* \xrightarrow{\delta_2} Q &\quad \wedge \quad \sigma = \delta_1 + \delta_2 \ \vee \\ P \xrightarrow{\delta_1}\xrightarrow{\sigma}\xrightarrow{\delta_2} Q &\quad \wedge \quad \sigma \in \overline{A}, \delta_1, \delta_2 \in \mathbb{R} \end{aligned}$$

exists in R then $P \xrightarrow{\sigma} Q$ also exists in R.

The output-δ-τ-closure of a DLTS transition relation $R \subseteq (S \times (Act \cup \mathbb{R}) \times S)$, is the relation R' such that

1. *R' is output-δ-τ-transitive.*
2. *$R' \supseteq R$.*
3. *For any output-δ-τ-transitive relation R'', $R'' \supseteq R \Rightarrow R'' \supseteq R'$.*

Transitions from state P to state Q by action σ in Output-δ-τ-closure are denoted by $P \overset{\sigma}{\Longrightarrow}_o Q$. The predicate $P \overset{\sigma}{\Longrightarrow}_o$ is true when there is at least one transition from state P via action $\sigma \in Act \cup \mathbb{R}$. Output-$\delta$-$\tau$-closure models time-and-tau-abstracted output actions; i.e., the closure relation has additional output transitions when they occur in conjunction with time-passing or internal-actions. Input, τ, and δ actions are not time-abstracted, only tau-abstracted. Output-δ-τ-closure extends implementation transition relations to match the specification output behaviors, but it does not allow the timing of inputs, δ's, or τ's to vary.

In addition to the closures, the following two projections are defined. They are subsets of the DLTS transition relation $\longrightarrow$. They define the sets of specification and implementation time-passing actions that must be subsets of each other's time actions—i.e., the $\xrightarrow{\delta}$ transitions leading to τ's and inputs or τ's and outputs respectively.

Definition 26. Input Projection: $\dashrightarrow_i$.

$$\dashrightarrow_i \subseteq S \times \mathbb{R} \times S \triangleq$$

$$\{\langle\langle l, \pi_i\rangle, \delta, \langle l, \pi_j\rangle\rangle \mid \langle\langle l, \pi_i\rangle, \delta, \langle l, \pi_j\rangle\rangle \in \longrightarrow \wedge \exists \langle\langle l, \pi_k\rangle, \alpha, \_\rangle \in \longrightarrow [\pi_i \leq \pi_k \wedge \pi_j \leq \pi_k \wedge \alpha \in \mathcal{A} \cup \{\tau\}]\}$$

Definition 27. Output Projection: $\dashrightarrow_o$.

$$\dashrightarrow_o \subseteq S \times \mathbb{R} \times S \triangleq$$

$$\{\langle\langle l, \pi_i\rangle, \delta, \langle l, \pi_j\rangle\rangle \mid \langle\langle l, \pi_i\rangle, \delta, \langle l, \pi_j\rangle\rangle \in \longrightarrow \wedge \exists \langle\langle l, \pi_k\rangle, \beta, \_\rangle \in \longrightarrow [\pi_i \leq \pi_k \wedge \pi_j \leq \pi_k \wedge \beta \in \bar{\mathcal{A}} \cup \{\tau\}]\}$$

The following example illustrates projections. If $X \xrightarrow{3} X' \xrightarrow{a} X''$, and only input a is possible from X', then $X \overset{3}{\dashrightarrow}_i X'$, but $X \not\overset{3}{\dashrightarrow}_o X'$. However, if $X \xrightarrow{3} X' \xrightarrow{\tau} X''$, then $X \overset{3}{\dashrightarrow}_i X'$ and $X \overset{3}{\dashrightarrow}_o X'$.

Next, a predicate that allows implementation outputs to have tighter upper-bounds than specification outputs is defined. It also relaxes the superset relationship between implementation and specification inputs when simultaneous inputs and outputs are possible from the same location. i.e., output timing constraints take precedence over input timing constraints when they conflict. This is reasonable, because when the implementation must perform an output, it causes it to happen. After that, whether or not the implementation's input behaviors satisfy the specification's is determined by the TLC relation of the post-output to-locations.

Definition 28. Output-Bound: *ob*. $ob : S_I \times \mathbb{R} \times S_S \times \wp(\langle S_I \times S_S\rangle) \longrightarrow \{t, f\} \triangleq$

$$ob(I, \delta, S, \mathcal{R}) \Leftrightarrow$$

$$I \overset{\delta}{\not\Longrightarrow}_o \wedge \tag{18}$$

$$\exists\, \delta_1 \in \mathbb{R}, I' \in S_I, \beta \in \bar{A} \cup \{\tau\}[I \overset{\delta_1}{\Longrightarrow}_o I' \wedge I' \xrightarrow{\beta} \wedge \tag{19}$$

$$\forall\, \delta_2 \geq \delta_1, S' \in S_S, I'' \in S_I[S \xrightarrow{\delta_2} S' \Rightarrow (\langle I', S' \rangle \in \mathcal{R} \wedge \tag{20}$$

$$(I \xrightarrow{\delta_2} I'' \Rightarrow \langle I'', S' \rangle \in \mathcal{R}))]] \tag{21}$$

Conjunct 18 requires that the implementation cannot do δ. Conjunct 19 requires the implementation system to be constrained by a location invariant to produce an output or τ. Conjunct 20 ensures that future specification actions are matched by the implementation at the time it produces the output, and conjunct 21 specifies that there are no other future implementation locations that do not also match the specification's behavior (bisimulation).

Output-bound allows faster implementation outputs in TSA locations where both inputs and outputs are possible. For example, output-bound allows us to accept an *And* with output delays in [2,4] as an implementation of an *And* specification with delays [1,6]. Without output-bound, only the lower bound of a delay could change, and in this example, only an *And* implementation with an upper bound of 6 would satisfy TLC. Output-bound formalizes the notion that as long as an implementation's output occurs within the bounds of the same output in the specification, it can occur in accordance with a tighter location invariant even though the specification could remain in its location longer and subsequently accept future inputs. Without this exception, TLC generally cannot accept implementations with less output variation in locations where otherwise unconstrained inputs are also possible. Modeling locations with both inputs and outputs possible is important for accurate modeling of real systems as well as abstracting behavior into simpler machines with fewer locations.

Based on the preceding definitions, the partial order that temporally and behaviorally relaxes weak timed bisimulation is defined as follows.

Definition 29. $\mathcal{LC}^t$**.** *A binary relation* $\mathcal{LC}^t \subseteq S_I \times S_S$ *over DLTS automata states is a timed logic conformation between an implementation DLTS* $\langle S_I, Act_I, \longrightarrow_I, \langle l_0, \pi_0 \rangle_I \rangle$*, and specification DLTS* $\langle S_S, Act_S, \longrightarrow_S, \langle l_0, \pi_0 \rangle_S \rangle$*, iff*

$$\forall\ \langle I, S \rangle \in \mathcal{LC}^t, \alpha \in \mathcal{A}, \beta \in \bar{A} \cup \{\tau\}, \delta \in \mathbb{R}\ [$$

$$\begin{aligned}
\forall\ S'\ [S \xrightarrow{\alpha} S' &\ \ \Rightarrow\ \ \exists\ I'\ [I \overset{\alpha}{\Longrightarrow}_o I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]] \wedge && (22)\\
\forall\ S'\ [S \xrightarrow{\beta} S' &\ \ \Rightarrow\ \ \exists\ I'\ [I \overset{\widehat{\beta}}{\Longrightarrow}_o I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]] \wedge && (23)\\
\forall\ I'\ [I \xrightarrow{\alpha} I' \wedge S \overset{\alpha}{\Longrightarrow} &\ \ \Rightarrow\ \ \exists\ S'\ [S \overset{\alpha}{\Longrightarrow}_i S' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]] \wedge && (24)\\
\forall\ I'\ [I \xrightarrow{\beta} I' &\ \ \Rightarrow\ \ \exists\ S'\ [S \overset{\widehat{\beta}}{\Longrightarrow}_i S' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]] \wedge && (25)\\
\forall\ S'\ [S \overset{\delta}{\dashrightarrow}_i S' &\ \ \Rightarrow\ \ (\exists\ I'\ [I \overset{\delta}{\Longrightarrow}_o I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t] \vee ob(I, \delta, S, \mathcal{LC}^t))] \wedge && (26)\\
\forall\ I'\ [I \overset{\delta}{\dashrightarrow}_o I' &\ \ \Rightarrow\ \ \exists\ S'\ [S \overset{\delta}{\Longrightarrow}_i S' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]]] && (27)
\end{aligned}$$

Formulas 22 and 23 require the implementation to simulate the observable behaviors of the specification. The implementation has considerable freedom for matching the specification via the output-δ-τ-closure; it can match inputs (α) or internal specification actions ($\beta = \tau$) while doing its own internal actions; and it can pass time and/or execute internal actions of its own to match specification outputs ($\beta \neq \tau$). Formulas 24 and 25 require the specification to simulate observable behaviors of the implementation. Formula 24 weakens standard weak bisimulation by allowing implementations with **irrelevant inputs** (i.e., inputs that are not possible from specification state S) as long as there is a mapping from the relevant subset of the implementation's state space to the specification's state space, just as Stevens formalized without timing (Ste94). Formula 25 requires the specification to simulate *all* outputs and τ's of the implementation. Formula 26 ensures that all specification time derivatives leading to specification inputs or τ's (i.e., those deltas where $S \overset{\delta}{\dashrightarrow}_i$) are simulated by the implementation with output-bound exceptions allowed, and Formula 27 ensures that *all* implementation time derivatives leading to implementation outputs or τ's (i.e., those deltas where $I \overset{\delta}{\dashrightarrow}_o$) are simulated by the specification.

4.5 Timed Logic Conformance Example

TSA X and Y in Figure 6 serve to illustrate the $\mathcal{LC}^t$ relation. X and Y are annotated with intervals to help visualize their induced DLTS's.

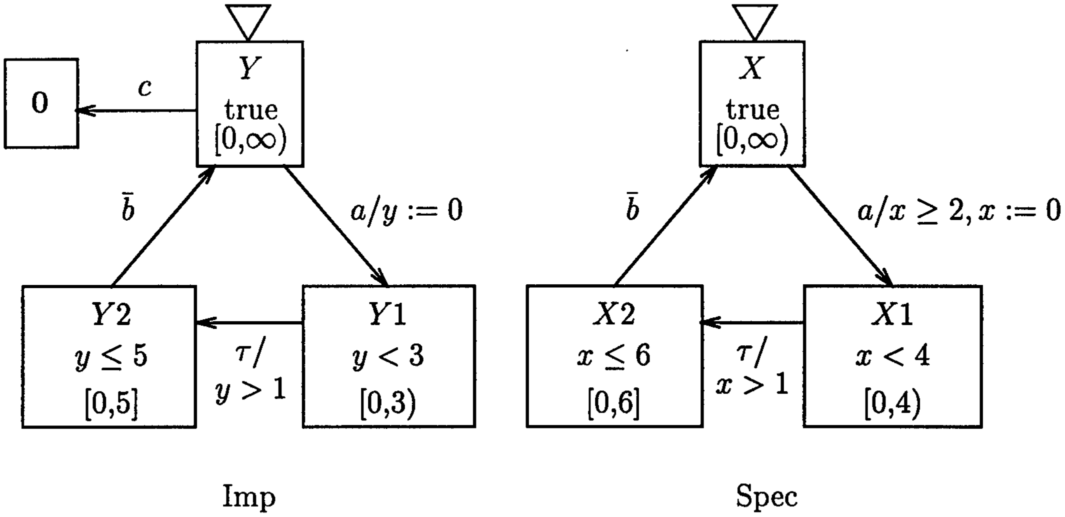


Figure 6. Simple $Y \;{}_{o}\underline{\diamond}_{i}\; X$ TSA.

The DLTS automata for X and Y have uncountable states, but the intervals annotated in the states of Figure 6 represent the value of the clocks x and y in X and Y states respectively. The following formulas prove that

$$\begin{aligned}\mathcal{LC}^t \;\triangleq\; & \{\langle Y_y, X_x\rangle \mid x, y \in \mathbb{R}\} \cup \\ & \{\langle Y1, X1\rangle \mid y \in [0,3) \wedge ((x \in [0,1] \wedge x \geq y) \vee (x \in (1,4) \wedge x \leq y+1))\} \cup \\ & \{\langle Y1_y, X2_x\rangle \mid y \in [0,3) \wedge x \in [0,4) \wedge x \leq y+1\} \cup\end{aligned}$$

$$\{\langle Y2_y, X1_x\rangle \mid y \in (0,5] \wedge x \in (1,4) \wedge x \leq y+1\} \cup$$

$$\{\langle Y2_y, X2_x\rangle \mid y \in [0,5] \wedge x \in [0,6] \wedge x \leq y+1\}$$

is a Timed Logic Conformation and that $\langle Y_0, X_0\rangle \in \mathcal{LC}^t$. Note that state-name subscripts represent the timed state via the value of the clock, and that the following formulas are numbered according to the corresponding formula number in Definition 29. Also note that formulas not shown are vacuously true (e.g., specification states without inputs vacuously satisfy Formula 22).

Relating states $\langle Y_y, X_x\rangle \forall\ x, y \in \mathbb{R}$

$$(22)\quad [X_x \xrightarrow{a} X1_0 \Rightarrow Y_x \xrightarrow{a} Y1_0 \wedge \langle Y1_0, X1_0\rangle \in \mathcal{LC}^t]$$

$$(24)\quad [(Y_y \xrightarrow{a} Y1_0 \wedge X_x \overset{a}{\Longrightarrow}) \ \Rightarrow X_x \xrightarrow{a} X1_0 \wedge \langle Y1_0, X1_0\rangle \in \mathcal{LC}^t]$$

$$(26)\quad [X_x \overset{\delta}{\dashrightarrow}_i X_{x+\delta} \Rightarrow Y_y \xrightarrow{\delta} Y_{y+\delta} \wedge \langle Y_{y+\delta}, X_{x+\delta}\rangle \in \mathcal{LC}^t]$$

Observe that Y's location $\mathbf{0}$ need not be associated with any corresponding location in X, since $X \overset{c}{\not\Longrightarrow}$ satisfies Formula 24 implication antecedent vacuously. If location 0 had derivatives, it would not matter that location 0 is not related to derivative locations of X; they are unreachable since the specification declares that c shall not occur. Note also that for $x \in [0,2), X_x \overset{a}{\not\Longrightarrow}$ satisfies Formula 24 vacuously, allowing $\langle Y_x, X_x\rangle$ to remain in $\mathcal{LC}^t$.

Note however, $\mathcal{LC}^t$ would be reduced to $\varnothing$ if $Y \overset{\bar{c}}{\to} \mathbf{0}$ instead of $Y \overset{c}{\to} \mathbf{0}$ because $\nexists_{\mathbf{0}} X \xrightarrow{\bar{c}} \mathbf{0}$ falsifies Formula 25, removing $\{\langle Y_y, X_x\rangle \mid x, y \in \mathbb{R}\}$ from $\mathcal{LC}^t$. Without $\{\langle Y_y, X_x\rangle \mid x, y \in \mathbb{R}\} \subseteq \mathcal{LC}^t$, $\langle Y2_y, X2_x\rangle$, $\langle Y1_y, X2_x\rangle$, $\langle Y2_y, X1_x\rangle$, $\langle Y1_y, X1_x\rangle$, cannot be in $\mathcal{LC}^t$ either, because the formulas require the to-locations to be in the relation, so $\mathcal{LC}^t = \varnothing$.

Relating states $\langle Y1, X1\rangle \forall\ y \in [0,3) \wedge ((x \in [0,1] \wedge x \geq y) \vee (x \in (1,4) \wedge x \leq y+1))$

$$(23)\quad \forall_{x\in[0,4),y\in[0,3)}\ [x \leq y+1 \wedge X1_x \xrightarrow{\tau} X2_x \Rightarrow Y1_y \overset{0}{\Longrightarrow}_o Y2_y \wedge \langle Y2_y, X2_x\rangle \in \mathcal{LC}^t]$$

(25) $\forall_{y \in [0,3), x \in (1,4)} [x \leq y + 1 \wedge Y1_y \xrightarrow{\tau} Y2_y \Rightarrow X1_x \overset{0}{\Longrightarrow}_i X2_x \wedge \langle Y2_y, X2_x \rangle \in \mathcal{LC}^t]$

(25) $\forall_{y \in [0,3), x \in [0,1]} [x \geq y \wedge Y1_y \overset{\tau}{\not\longrightarrow}]$

(26) $\forall_{x \in [0,4), y \in [0,3), \delta \in \mathbb{R}} [(x > y \vee x \leq y + 1) \wedge X1_x \overset{\delta}{\dashrightarrow}_i X1_{x+\delta} \Rightarrow$

$$(Y1_y \overset{\delta}{\Longrightarrow}_o Y1_{y+\delta} \wedge \langle Y1_{y+\delta}, X1_{x+\delta} \rangle \in \mathcal{LC}^t) \vee ob(Y1_y, \delta, X1_x, \mathcal{LC}^t)]$$

(27) $\forall_{y \in [0,3), x \in [0,1], \delta \in \mathbb{R}} [x \geq y \wedge Y1_y \xrightarrow{\delta} Y1_{y+\delta} \Rightarrow X1_x \overset{\delta}{\Longrightarrow}_i X1_{x+\delta} \wedge \langle Y1_{y+\delta}, X1_{x+\delta} \rangle \in \mathcal{LC}^t]$

(27) $\forall_{y \in [0,3), x \in [0,4), \delta \in \mathbb{R}} [(x > y \vee x \leq y + 1) \wedge Y1_y \overset{\delta}{\dashrightarrow}_o Y1_{y+\delta} \Rightarrow$

$$X1_x \overset{\delta}{\Longrightarrow}_i X1_{x+\delta} \wedge \langle Y1_{y+\delta}, X1_{x+\delta} \rangle \in \mathcal{LC}^t]$$

If $Y1 \xrightarrow{\tau}$'s guard were $y \geq 1$ then Formula 25 would be false at time $x = y = 1$ for implementation transitions $Y1_1 \xrightarrow{\tau} Y2_1$ because $\langle Y2_1, X1_1 \rangle \notin \mathcal{LC}^t$ and there is no $X1_1 \overset{0}{\Longrightarrow}_i X2_1$ transition in X. This would mean that Formula 27 is also false for $Y1_y \xrightarrow{\delta} Y1_2, y \in [0,1], \delta = 1 - y$ because $\langle Y1_1, X1_1 \rangle \notin \mathcal{LC}^t$. This removes $\langle Y1_y, X1_x \rangle$ from $\mathcal{LC}^t$. Consequently $\langle Y_y, X_x \rangle$, $\langle Y2_y, X2_x \rangle$, $\langle Y1_y, X2_x \rangle$, $\langle Y2_y, X1_x \rangle$, and the remaining $\langle Y1_y, X1_x \rangle$, cannot be in $\mathcal{LC}^t$ either, so $\mathcal{LC}^t = \varnothing$.

Relating states $\langle Y1_y, X2_x \rangle \mid y \in [0,3) \wedge x \in [0,4) \wedge x \leq y + 1$

(23) $[X2_x \xrightarrow{\bar{b}} X_x \Rightarrow Y1_y \overset{\bar{b}}{\Longrightarrow}_o Y_y \wedge \langle Y_y, X_x \rangle \in \mathcal{LC}^t]$

(25) $[Y1_y \xrightarrow{\tau} Y2_y \Rightarrow X2_x \overset{0}{\Longrightarrow}_i X2_x \wedge \langle Y2_y, X2_x \rangle \in \mathcal{LC}^t]$

(27) $\forall_{\delta \in \mathbb{R}} [Y1_y \overset{\delta}{\dashrightarrow}_o Y1_{y+\delta} \Rightarrow X2_x \overset{\delta}{\Longrightarrow}_i X2_{x+\delta} \wedge \langle Y1_{y+\delta}, X2_{x+\delta} \rangle \in \mathcal{LC}^t]$

Relating states $\langle Y2_y, X1_x \rangle \forall\ y \in (0,5] \wedge x \in (1,4) \wedge x \leq y + 1$

(23) $[X1_x \xrightarrow{\tau} X2_x \Rightarrow Y2_y \overset{0}{\Longrightarrow}_o Y2_y \wedge \langle Y2_y, X2_x \rangle \in \mathcal{LC}^t]$

(25) $[Y2_y \xrightarrow{\bar{b}} Y_y \Rightarrow X1_x \overset{\bar{b}}{\Longrightarrow}_i X_x \wedge \langle Y_y, X_x \rangle \in \mathcal{LC}^t]$

(27) $[Y2_y \overset{\delta}{\dashrightarrow}_o Y2_{y+\delta} \Rightarrow X1_x \overset{\delta}{\Longrightarrow}_i X1_{x+\delta} \wedge \langle Y2_{y+\delta}, X1_{x+\delta} \rangle \in \mathcal{LC}^t]$

Relating states $\langle Y2_y, X2_x\rangle \forall\ y \in [0,5] \wedge x \in [0,6] \wedge x \leq y+1$

$$(23)\quad [X2_x \xrightarrow{\bar{b}} X_x \Rightarrow Y2_y \xrightarrow{\bar{b}} Y_y \wedge \langle Y_y, X_x\rangle \in \mathcal{LC}^t]$$

$$(25)\quad [Y2_y \xrightarrow{\bar{b}} Y_y \Rightarrow X2_x \xrightarrow{\bar{b}} X_x \wedge \langle Y_y, X_x\rangle \in \mathcal{LC}^t]$$

$$(27)\quad \forall_{\delta\in\mathbb{R}}\, [Y2_y \overset{\delta}{\dashrightarrow}_o Y2_{y+\delta} \Rightarrow X2_x \overset{\delta}{\Longrightarrow}_i X2_{x+\delta} \wedge \langle Y2_{y+\delta}, X2_{x+\delta}\rangle \in \mathcal{LC}^t]$$

4.6 Comparing TLC to Other Relations

Theoretically, TLC's relationship with other formal equivalence, partial order, and refinement relations like TCCS's weak timed bisimulation, CTR, and timed simulation is important to understand. Since TLC is asymmetrically defined over a different formalism, comparing them is not generally rigorously possible. One can see that TLC is weaker than TCCS weak timed bisimulation because TLC ignores temporal differences between actions. TLC is not comparable in a formal sense to CTR's refinement relation because they formalize the relationship between implementation and specification inputs in opposite directions; i.e., for CTR agents $X \triangleq a.[1,5].b.X$ and $Y \triangleq a.[2,4].b.Y$ $Y \trianglelefteq X$ and $X \ntrianglelefteq Y$, but the opposite is true for TLC: $X\ {}_o\!\succeq_i Y$ and $Y\ {}_o\!\not\succeq_i X$. Comparing timed simulation to TLC raises similar issues. The fact that TLC allows constrained inputs that violate timed simulation's nonblocking requirement makes TLC weaker than timed simulation in that situation. However, when used on models that define all inputs in all states for all times, TLC is stronger in the sense that it requires the set of output variables of the two processes to be the same, while timed simulation does not. Other than saying "TLC is weaker than weak timed bisimulation," about all that can be said is "TLC is different from the rest."

4.7 Properties of Timed Logic Conformance

In the interest of applying TLC to a hierarchical design process, it must be shown to show that $\mathcal{LC}^t$ induces a partially ordered binary relation (reflexive, antisymmetric, and transitive) over the set of DLTS automata induced from the set of TSA. Reflexivity is an important property for design purposes, because it must always be possible to substitute a component for itself. Antisymmetry is likewise important because models that can be substituted for each other are "equivalent" or the same. Transitivity is the property that guarantees hierarchical verification of the $\mathcal{LC}^t$ relation. Unfortunately, without restricting TSA modeling, TLC is not transitive over the induced DLTS. The following TSA modeling constraints are required to preserve transitivity:

Definition 30. TSA Modeling Constraints.

1. *A location has a location invariant iff it has one or more output or τ transitions from it. This is similar to TCCS's persistence property, but only for outputs and τ's.*
2. *No initial location has an invariant (all initial locations are stable). Hence, all TSA must receive at least one input before generating an output.*
3. *No output or τ transition is guarded by an upper-bound stronger than the from-location invariant.*
4. *No to-location of a transition has a stronger location invariant than the from-location unless the clocks involved in the strengthened invariants are reset.*

These are reasonable modeling constraints—especially for the hardware domain where devices control the timing of their outputs but not their inputs. The constraints strengthen the causal relationship of the models and their outputs and they increase the fidelity between the models and the physical devices they represent. The first modeling constraint increases fidelity because devices that are not broken cannot take an indefinite amount of time to produce an output. It also prohibits the situation where a receiving device "forces" an input to occur by an expiring location invariant. Note that a model can still place an upper-bound on an input, but the upper-bound constraint for inputs may only be expressed by a guard, not a location invariant. Hence an upper-bounded input guard can disable the input, but it cannot *cause* the input to occur. The second modeling constraint increases fidelity by modeling the situation where all circuits must have logic that initializes them to a known state to be able to rely on the correct function of the device. Devices like clocks are modeled by an initial state with a reset input transition before repeatedly toggling the clock output. The third and fourth modeling constraints avoid locations that prohibit the passing of time (i.e., locations that make the TSA Zeno). Constraints 3 and 4 prohibit Zeno locations with no enabled transition as the location invariant expires.

Since parallel composition rules Com2 and Com3 relate inputs and outputs together, tighter inputting component constraints can adversely constrain outputting component timing and violate the fourth modeling constraint. To ensure that compositions continue to satisfy the modeling constraints and that TLC is transitive for compositions, a condition on compositions that specifies the necessary timing relationship between inputting and outputting components must be imposed.

A parallel composition with an output offered in a non-accepting location is a design error called **computation interference** (CI). The following property excludes CI from compositions.

Definition 31. CI-free Parallel Composition.

1. All non-parallel TSA and their induced DLTS are CI-free.

2. The n-parallel TSA $\mathcal{T}_p = \langle \mathcal{L}_p, Act_p, \Xi_p, \langle\langle l_{0\,1}, \rho_{0\,1}\rangle, \langle l_{0\,2}, \rho_{0\,2}\rangle, \cdots, \langle l_{0\,n}, \rho_{0\,n}\rangle\rangle, \longmapsto_p\rangle$, *and its induced DLTS automaton* $\mathcal{D} = \langle S_p, Act_p, \longrightarrow_p, \langle\langle l_{0\,1}, \vec{0}_1\rangle, \langle l_{0\,2}, \vec{0}_2\rangle, \cdots, \langle l_{0\,n}, \vec{0}_n\rangle\rangle\rangle$, *are CI-free over location:state space* $L\colon S\ L \subseteq \mathcal{L}_p, S \subseteq S_p$ *when:*

$$\forall\ \langle\langle l_1, \rho_{l\,1}\rangle, \langle l_2, \rho_{l\,2}\rangle, \cdots, \langle l_n, \rho_{l\,n}\rangle\rangle : \langle\langle l_1, \vec{\pi}_1\rangle, \langle l_2, \vec{\pi}_2\rangle, \cdots, \langle l_n, \vec{\pi}_n\rangle\rangle \in L\colon S[$$

$$\forall\ \langle l_i, \rho_{l\,i}\rangle \overset{\overline{\alpha},\rho_i,\eta_i}{\longmapsto}_i \langle l'_i, \rho'_{l\,i}\rangle[(\vec{\pi}_i \in \rho_i \wedge \vec{\pi}_i \in \rho_{l\,i} \wedge \vec{\pi}_i[\eta_i := 0] \in \rho'_{l\,i}) \Rightarrow \tag{28}$$

$$\forall\ 1 \leq j \leq n[(j \neq i \wedge \alpha \in \mathcal{A}_j) \Rightarrow$$

$$\exists\ \langle l_j, \rho_{l\,j}\rangle \overset{\alpha,\rho_j,\eta_j}{\longmapsto}_j \langle l'_j, \rho'_{l\,j}\rangle[\vec{\pi}_j \in \rho_j \wedge \vec{\pi}_j \in \rho_{l\,j} \wedge \vec{\pi}_j[\eta_j := 0] \in \rho'_{l\,j}]]] \wedge$$

$$\forall\ \langle l_i, \vec{\pi}_i\rangle \overset{\delta}{\dashrightarrow}_{oi}\ \langle l'_i, \vec{\pi}'_i\rangle, 1 \leq j \leq n[j \neq i \Rightarrow \exists\ \langle l_j, \vec{\pi}_j\rangle \overset{\delta}{\Longrightarrow}_j \langle l'_j, \vec{\pi}'_j\rangle]] \tag{29}$$

In words, a composition is CI-free when all composition receivers accept every output offered by transmitters in the composition's reachable state space (Formula 28) and no agent in the composition prohibits the passing of time until an output occurs (Formula 29). All non-parallel TSA are CI-free by definition. A top-level parallel-composed specification must be CI-free over its own reachable state space.

With the modeling constraints and the CI-free property, the TLC partial-order properties can be established. For the purposes of the following proofs and definitions, assume that each subscripted $\mathcal{LC}^t$ relation $\mathcal{LC}^t_i \subset S_I \times S_S$ is a timed logic conformation according to Def. 29.

Lemma 2 *The relation*

$$Id \triangleq \{\langle x, x\rangle \mid x \in S\} \tag{30}$$

is a timed logic conformation.

Proof

Given any DLTS Automaton $\langle S, Act, \longrightarrow, s_0\rangle, P \in S, \gamma \in Act \cup \mathbb{R}$, every transition $P \xrightarrow{\gamma} P' \in\longrightarrow$ can be matched by itself in the superset transition relations $P \overset{\gamma}{\Longrightarrow}_i P'$, and $P \overset{\gamma}{\Longrightarrow}_o P'$; and for $P \xrightarrow{\tau} P' \in\longrightarrow$ since $P \xrightarrow{\tau} P' \Rightarrow (P \xrightarrow{0} P \xrightarrow{\tau} P' \xrightarrow{0} P') \Rightarrow (P \overset{0}{\Longrightarrow} P \overset{\tau}{\Longrightarrow} P' \overset{0}{\Longrightarrow} P') \Rightarrow P \overset{0}{\Longrightarrow} P' \Rightarrow P \overset{\hat{\tau}}{\Longrightarrow} P'$, τ-closure ensures $P \overset{\hat{\tau}}{\Longrightarrow}_i P'$, and $P \overset{\hat{\tau}}{\Longrightarrow}_o P'$, therefore, $\langle P, P\rangle \in Id \Rightarrow (P \overset{\hat{\gamma}}{\Longrightarrow}_i P' \wedge P \overset{\hat{\gamma}}{\Longrightarrow}_o P' \wedge \langle P', P'\rangle \in Id)$ therefore Formulas 22 through 27 are satisfied and Id is a timed logic conformation. □

Lemma 3 *The composition relation*

$$\mathcal{LC}^t_1\mathcal{LC}^t_2 \triangleq \{\langle p, r\rangle \mid \langle p, q\rangle \in \mathcal{LC}^t_1 \wedge \langle q, r\rangle \in \mathcal{LC}^t_2\} \tag{31}$$

is a timed logic conformation.

Proof

The proof proceeds by assuming that $\langle P, R\rangle \in \mathcal{LC}_1^t\mathcal{LC}_2^t$ and showing that for all possible actions that must be matched in each of the Formulas 22 through 27 in Def. 29, the actions are matched across the composition and $\langle P', R'\rangle \in \mathcal{LC}_1^t\mathcal{LC}_2^t$.

1. Formulas 22 and 23: Since $\mathcal{LC}_2^t$ is a logic conformation, according to Formulas 22 and 23, every $\sigma \in Act, R' \in S_R$, $R \xrightarrow{\sigma} R'$ is matched by some Q' such that $Q \overset{\hat{\sigma}}{\Longrightarrow}_o Q' \wedge \langle Q', R'\rangle \in \mathcal{LC}_2^t$. Since $Q \overset{\hat{\sigma}}{\Longrightarrow}_o Q'$ is not $Q \xrightarrow{\sigma} Q'$, a Lemma of the following form is needed:

   $$\forall\ \sigma \in Act[(\langle P, Q\rangle \in \mathcal{LC}_1^t \wedge Q \overset{\hat{\sigma}}{\Longrightarrow}_o Q') \Rightarrow (P \overset{\hat{\sigma}}{\Longrightarrow}_o P' \wedge \langle P', Q'\rangle \in \mathcal{LC}_1^t)]$$

   The proof of this is deferred to Lemma 4. Applying Lemma 4, $Q \overset{\hat{\sigma}}{\Longrightarrow}_o Q' \Rightarrow P \overset{\hat{\sigma}}{\Longrightarrow}_o P' \wedge \langle P', Q'\rangle \in \mathcal{LC}_1^t$ and by the definition of the composition relation $\langle P', R'\rangle \in \mathcal{LC}_1^t\mathcal{LC}_2^t$.

2. Formula 24: Since $\mathcal{LC}_1^t$ is a logic conformation, according to Formula 24, $\forall\ \alpha \in \mathcal{A}, P' \in S_P[P \xrightarrow{\alpha} P' \wedge Q \overset{\alpha}{\Longrightarrow}$ is matched by some Q' such that $Q \overset{\alpha}{\Longrightarrow}_i Q' \wedge \langle P', Q'\rangle \in \mathcal{LC}_1^t]$. Since $Q \overset{\alpha}{\Longrightarrow}_i Q'$ is not $Q \xrightarrow{\alpha} Q'$, a Lemma of the following form is needed:

   $$\forall\ \alpha \in \mathcal{A}[(\langle Q, R\rangle \in \mathcal{LC}_2^t \wedge R \overset{\alpha}{\Longrightarrow} \wedge Q \overset{\alpha}{\Longrightarrow}_i Q') \Rightarrow (R \overset{\alpha}{\Longrightarrow}_i R' \wedge \langle Q', R'\rangle \in \mathcal{LC}_2^t)]$$

   The proof of this is deferred to Lemma 5.

   (a) $R \overset{\alpha}{\Longrightarrow}$: Applying Lemma 5 implies $R \overset{\alpha}{\Longrightarrow}_i R' \wedge \langle Q', R'\rangle \in \mathcal{LC}_2^t$, and by the definition of the composition relation $\langle P', R'\rangle \in \mathcal{LC}_1^t\mathcal{LC}_2^t$.

   (b) $R \overset{\alpha}{\not\Longrightarrow}$: R need not match Q's α and $\langle P', R'\rangle$ is not required in $\mathcal{LC}_1^t\mathcal{LC}_2^t$.

3. Formula 25: Since $\mathcal{LC}_1^t$ is a logic conformation, according to Formula 25, $\forall\ \beta \in \overline{\mathcal{A}} \cup \{\tau\}, P' \in S_P[P \xrightarrow{\beta} P'$ is matched by some Q' such that $Q \overset{\hat{\beta}}{\Longrightarrow}_i Q' \wedge \langle P', Q'\rangle \in \mathcal{LC}_1^t]$. Since $Q \overset{\hat{\beta}}{\Longrightarrow}_i Q'$

is not $Q \xrightarrow{\beta} Q'$, a Lemma of the following form is needed:

$$\forall\, \beta \in \overline{\mathcal{A}} \cup \{\tau\}[(\langle Q,R\rangle \in \mathcal{LC}_2^t \wedge Q \overset{\widehat{\beta}}{\Longrightarrow}_i Q') \Rightarrow (R \overset{\widehat{\beta}}{\Longrightarrow}_i R' \wedge \langle Q',R'\rangle \in \mathcal{LC}_2^t)]$$

The proof of this is deferred to Lemma 6. Applying Lemma 6 implies $R \overset{\widehat{\beta}}{\Longrightarrow}_i R' \wedge \langle Q',R'\rangle \in \mathcal{LC}_2^t$, and by the definition of the composition relation $\langle P',R'\rangle \in \mathcal{LC}_1^t\mathcal{LC}_2^t$.

4. Formula 26: Since $\mathcal{LC}_2^t$ is a logic conformation, according to Formula 26, every $\delta \in \mathbb{R}, R' \in S_R$, $R \overset{\delta}{\dashrightarrow}_i R'$ is matched by some Q' such that $Q \overset{\delta}{\Longrightarrow}_o Q' \wedge \langle Q',R'\rangle \in \mathcal{LC}_2^t$ or the predicate $ob(Q,\delta,R,\mathcal{LC}_2^t)$ is true.

   (a) $Q \overset{\delta}{\Longrightarrow}_o Q' \wedge \langle Q',R'\rangle \in \mathcal{LC}_2^t$: Since $Q \overset{\delta}{\Longrightarrow}_o Q'$ is not $Q \overset{\delta}{\dashrightarrow}_i Q'$, a Lemma of the following form is needed:

$$\begin{aligned}\forall\, \delta \in \mathbb{R} \quad & [(Q \overset{\delta}{\Longrightarrow}_o Q' \wedge \exists\, \alpha \in \mathcal{A}[Q' \overset{\alpha}{\Longrightarrow}_o]) \Rightarrow \\ & (P \overset{\delta}{\Longrightarrow}_o P' \wedge \langle P',Q'\rangle \in \mathcal{LC}_1^t \vee ob(P,\delta,Q,\mathcal{LC}_1^t))]\end{aligned}$$

   The proof of this is deferred to Lemma 7. Applying Lemma 7, $\exists\, \alpha \in \mathcal{A}[Q' \overset{\alpha}{\Longrightarrow}_o]$, because Def. 26 requires that $R' \overset{\alpha}{\Longrightarrow}_o$, and Q' must match R's future inputs, so $(Q \overset{\delta}{\Longrightarrow}_o Q') \Rightarrow ((P \overset{\delta}{\Longrightarrow}_o P' \wedge \langle P',Q'\rangle \in \mathcal{LC}_1^t) \vee ob(P,\delta,Q,\mathcal{LC}_1^t))$.

      i. $P \overset{\delta}{\Longrightarrow}_o P' \wedge \langle P',Q'\rangle \in \mathcal{LC}_1^t$: By the definition of the composition, $\langle P',R'\rangle \in \mathcal{LC}_1^t\mathcal{LC}_2^t$.

      ii. $ob(P,\delta,Q,\mathcal{LC}_1^t)$: Then $ob(P,\delta,R,\mathcal{LC}_1^t\mathcal{LC}_2^t)$ from Def. 28 and the definition of the composition.

   (b) $ob(Q,\delta,R,\mathcal{LC}_2^t)$: From Def. 28, for some $\delta_1 < \delta$, $Q \overset{\delta_1}{\Longrightarrow}_o Q'$, and Formula 26 implies $(P \overset{\delta_1}{\Longrightarrow}_o P' \wedge \langle P',Q'\rangle \in \mathcal{LC}_1^t) \vee ob(P,\delta_1,Q,\mathcal{LC}_1^t)$.

      i. $P \overset{\delta_1}{\Longrightarrow}_o P' \wedge \langle P',Q'\rangle \in \mathcal{LC}_1^t$: Since $\delta_1 < \delta$ and P' conforms to every Q' reached by every $\delta > \delta_1$ and those Q' logically conform to every R' reached by those same δ, $ob(P,\delta,R,\mathcal{LC}_1^t\mathcal{LC}_2^t)$ holds by Def. 28 and the definition of the composition.

ii. $ob(P, \delta_1, Q, \mathcal{LC}_1^t)$: Since $\delta > \delta_1$, it is true that $ob(P, \delta, R, \mathcal{LC}_1^t\mathcal{LC}_2^t)$ from Def. 28 and the definition of the composition.

5. Formula 27: Since $\mathcal{LC}_1^t$ is a logic conformation, according to Formula 27, $\forall\, \delta \in \mathbb{R}, P' \in S_P[P \overset{\delta}{\dashrightarrow}_o P'$ is matched by some Q' such that $Q \overset{\delta}{\Longrightarrow}_i Q' \wedge \langle P', Q' \rangle \in \mathcal{LC}_1^t]$. Since $Q \overset{\delta}{\Longrightarrow}_i Q'$ is not $Q \overset{\delta}{\dashrightarrow}_o Q'$, a Lemma of the following form is needed:

$$\forall\, \delta \in \mathbb{R}[(\langle Q, R \rangle \in \mathcal{LC}_2^t \wedge Q \overset{\delta}{\Longrightarrow}_i Q' \wedge \exists\, \beta \in \overline{\mathcal{A}} \cup \{\tau\}[Q' \overset{\beta}{\Longrightarrow}_i]) \Rightarrow (R \overset{\delta}{\Longrightarrow}_i R' \wedge \langle Q', R' \rangle \in \mathcal{LC}_2^t)]$$

The proof of this is deferred to Lemma 8. Since Q is required to match the outputs of P, it is true that $\exists\, \beta \in \overline{\mathcal{A}} \cup \{\tau\}[Q' \overset{\beta}{\Longrightarrow}_i]$, and applying Lemma 8 implies $R \overset{\delta}{\Longrightarrow}_i R' \wedge \langle Q', R' \rangle \in \mathcal{LC}_2^t$, and by the definition of the composition relation $\langle P', R' \rangle \in \mathcal{LC}_1^t\mathcal{LC}_2^t$. □

Lemma 4 *Given that $\mathcal{LC}^t$ is a timed logic conformance:*

$$\forall\, \sigma \in Act[(\langle I, S \rangle \in \mathcal{LC}^t \wedge S \overset{\widehat{\sigma}}{\Longrightarrow}_o S') \Rightarrow (I \overset{\widehat{\sigma}}{\Longrightarrow}_o I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t)]$$

Proof

1. Case $S(\xrightarrow{\tau})^* S^m \xrightarrow{\sigma} S^n(\xrightarrow{\tau})^* S'$ (Def. 25, first disjunct):

   Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 23, every S^1, $S \xrightarrow{\tau} S^1$ is matched by some I^1 such that $I \overset{\widehat{\tau}}{\Longrightarrow}_o I^1 \wedge \langle I^1, S^1 \rangle \in \mathcal{LC}^t$. This is true inductively for more $\tau \in \tau^*$ and $S^1 \cdots S^m$ and $I^1 \cdots I^m$ according to Formula 23.

   According to Formulas 22 and 23, $S^m \xrightarrow{\sigma} S^n$ is matched by some I^n such that $I^m \overset{\widehat{\sigma}}{\Longrightarrow}_o I^n \wedge \langle I^n, S^n \rangle \in \mathcal{LC}^t$. When $S^m \xrightarrow{\widehat{\sigma}} S^n$ is $S^m \xrightarrow{0} S^n$, Formula 26 applies, and since every DLTS state has a $\delta = 0$ transition by definition, $I^m \overset{0}{\Longrightarrow}_o I^n \wedge \langle I^n, S^n \rangle \in \mathcal{LC}^t$.

   Finally, by Formula 23 any sequence of τ's in $S^n \xrightarrow{\tau^*} S'$ is matched by $I^n \xrightarrow{\widehat{\tau^*}} I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t$. And, by Def. 25, $(I \overset{\widehat{\tau^*}}{\Longrightarrow}_o I^m \overset{\widehat{\sigma}}{\Longrightarrow}_o I^n \overset{\widehat{\tau^*}}{\Longrightarrow}_o I') \Rightarrow I \overset{\widehat{\sigma}}{\Longrightarrow}_o I'$

2. Case $S \xrightarrow{\delta_1} S^m \xrightarrow{\sigma} S^n \xrightarrow{\delta_2} S' \wedge \sigma \in \overline{\mathcal{A}}, \delta_1, \delta_2 \in \mathbb{R}$ (Def. 25, third disjunct):

   (a) $I \overset{\delta_1}{\Longrightarrow}_o$:

      i. $S \overset{\delta_1}{\dashrightarrow}_i S^m$: Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 26, $S \overset{\delta_1}{\dashrightarrow}_i S^m$ is matched by some I^m such that $I \overset{\delta_1}{\Longrightarrow}_o I^m \wedge \langle I^m, S^m \rangle \in \mathcal{LC}^t$. Since $\langle I^m, S^m \rangle \in \mathcal{LC}^t \wedge S^m \xrightarrow{\sigma} S^n$, Formulas 22 and 23 require some I^n such that $I^m \overset{\hat{\sigma}}{\Longrightarrow}_o I^n \wedge \langle I^n, S^n \rangle \in \mathcal{LC}^t$.

         A. $I^n \overset{\delta_2}{\Longrightarrow}_o$:

         Case $S^n \overset{\delta_2}{\dashrightarrow}_i S'$: Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 26, $S^n \overset{\delta_2}{\dashrightarrow}_i S'$ is matched by some I' such that $I^n \overset{\delta_2}{\Longrightarrow}_o I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t$. And $(I \overset{\delta_1}{\Longrightarrow}_o I^m \overset{\hat{\sigma}}{\Longrightarrow}_o I^n \overset{\delta_2}{\Longrightarrow}_o I') \Rightarrow I \overset{\sigma}{\Longrightarrow}_o I'$ by Def. 25.

         Case $S^n \overset{\delta_2}{\not\dashrightarrow}_i S'$: I^n need not match S^n's δ_2 and $\langle I', S' \rangle$ is not required in $\mathcal{LC}^t$.

         B. $I^n \overset{\delta_2}{\not\Longrightarrow}_o$, $(ob(I^n, \delta_2, S^n, \mathcal{LC}^t)$ holds): Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 26, and Def. 28, $\exists\ I' \in S_I, \delta_2' \in \mathbb{R}[I^n \overset{\delta_2'}{\Longrightarrow}_o I' \wedge S^n \xrightarrow{\delta_2} S' \Rightarrow \langle I', S' \rangle \in \mathcal{LC}^t]$ And $(I \overset{\delta_1}{\Longrightarrow}_o I^m \overset{\hat{\sigma}}{\Longrightarrow}_o I^n \overset{\delta_2'}{\Longrightarrow}_o I') \Rightarrow I \overset{\sigma}{\Longrightarrow}_o I'$ by Def. 25.

      ii. $S \overset{\delta_1}{\not\dashrightarrow}_i S^m$ then I need not match S's δ_1 and $\langle I^m, S^m \rangle$ is not required in $\mathcal{LC}^t$.

   (b) $I \overset{\delta_1}{\not\Longrightarrow}_o$, $(ob(I, \delta_1, S, \mathcal{LC}^t)$ holds): Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 26, and the Def. 28, $\exists\ I^m \in S_I, \delta_1' \in \mathbb{R}[I \overset{\delta_1'}{\Longrightarrow}_o I^m \wedge S \xrightarrow{\delta_1} S^m \Rightarrow \langle I^m, S^m \rangle \in \mathcal{LC}^t]$. Since $\langle I^m, S^m \rangle \in \mathcal{LC}^t \wedge S^m \xrightarrow{\sigma} S^n$, Formulas 22 and 23 require some I^n such that $I^m \overset{\hat{\sigma}}{\Longrightarrow}_o I^n \wedge \langle I^n, S^n \rangle \in \mathcal{LC}^t$.

      i. $I^n \overset{\delta_2}{\Longrightarrow}_o$:

         A. $S^n \overset{\delta_2}{\dashrightarrow}_i S'$: Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 26, $S^n \overset{\delta_2}{\dashrightarrow}_i S'$ is matched by some I' such that $I^n \overset{\delta_2}{\Longrightarrow}_o I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t$. And $(I \overset{\delta_1'}{\Longrightarrow}_o I^m \overset{\hat{\sigma}}{\Longrightarrow}_o I^n \overset{\delta_2}{\Longrightarrow}_o I') \Rightarrow I \overset{\sigma}{\Longrightarrow}_o I'$ by Def. 25.

         B. $S^n \overset{\delta_2}{\not\dashrightarrow}_i S'$: I^n need not match S^n's δ_2 and $\langle I', S' \rangle$ is not required in $\mathcal{LC}^t$.

ii. $I^n \overset{\delta_2}{\not\Longrightarrow}_o$, ($ob(I^n, \delta_2, S^n, \mathcal{LC}^t)$ holds): Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 26, and the Def. 28, $\exists\ I' \in S_I, \delta_2' \in \mathbb{R}[I^n \overset{\delta_2'}{\Longrightarrow}_o I' \wedge S^n \xrightarrow{\delta_2} S' \Rightarrow \langle I', S'\rangle \in \mathcal{LC}^t]$ And $(I \overset{\delta_1'}{\Longrightarrow}_o I^m \overset{\widehat{\sigma}}{\Longrightarrow}_o I^n \overset{\delta_2'}{\Longrightarrow}_o I') \Rightarrow I \overset{\sigma}{\Longrightarrow}_o I'$ by Def. 25.□

Lemma 5 *Given that $\mathcal{LC}^t$ is a timed logic conformance:*

$$\forall\ \alpha \in \mathcal{A}, I' \in S_I[(\langle I, S\rangle \in \mathcal{LC}^t \wedge S \overset{\alpha}{\Longrightarrow} \wedge I \overset{\alpha}{\Longrightarrow}_i I') \Rightarrow \exists\ S' \in S_S[S \overset{\alpha}{\Longrightarrow}_i S' \wedge \langle I', S'\rangle \in \mathcal{LC}^t]]$$

Proof

1. $I(\xrightarrow{\tau})^* I^m \xrightarrow{\alpha} I^n(\xrightarrow{\tau})^* I'$ (from Def. 24, first disjunct): Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 25, every I^1, $I \xrightarrow{\tau} I^1$ is matched by some S^1 such that $S \overset{\widehat{\tau}}{\Longrightarrow}_i S^1 \wedge \langle I^1, S^1\rangle \in \mathcal{LC}^t$. This is true inductively for more $\tau \in \tau^*$ and $I^1 \cdots I^m$ and $S^1 \cdots S^m$ according to Formula 25. According to Formula 24, $I^m \xrightarrow{\alpha} I^n$ is matched by some S^n such that $S^m \overset{\alpha}{\Longrightarrow}_i S^n \wedge \langle I^n, S^n\rangle \in \mathcal{LC}^t$. And finally, by Formula 25 any sequence of τ's in $I^n \xrightarrow{\tau^*} I'$ is matched by $S^n \xrightarrow{\widehat{\tau^*}} S' \wedge \langle I', S'\rangle \in \mathcal{LC}^t$. And, by Def. 24, $(S \overset{\widehat{\tau^*}}{\Longrightarrow}_i S^m \overset{\sigma}{\Longrightarrow}_i S^n \overset{\widehat{\tau^*}}{\Longrightarrow}_i S') \Rightarrow S \overset{\sigma}{\Longrightarrow}_i S'$.

2. $I \xrightarrow{\delta_1} I^m \xrightarrow{\alpha} I^n \xrightarrow{\delta_2} I' \wedge \alpha \in \mathcal{A}, \delta_1, \delta_2 \in \mathbb{R}$ (Def. 24, third disjunct):

   (a) $I \overset{\delta_1}{\dashrightarrow}_o I^m$: According to Formula 27, $I \overset{\delta_1}{\dashrightarrow}_o I^m$ is matched by some S^m such that $S \overset{\delta_1}{\Longrightarrow}_i S^m \wedge \langle I^m, S^m\rangle \in \mathcal{LC}^t$. Since $\langle I^m, S^m\rangle \in \mathcal{LC}^t \wedge I^m \xrightarrow{\alpha} I^n$, Formula 24 requires some S^n such that $S^m \overset{\alpha}{\Longrightarrow}_i S^n \wedge \langle I^n, S^n\rangle \in \mathcal{LC}^t$.

      i. $I^n \overset{\delta_2}{\dashrightarrow}_o I'$: According to Formula 27, $I^n \overset{\delta_2}{\dashrightarrow}_o I'$ is matched by some S' such that $S^n \overset{\delta_2}{\Longrightarrow}_i S' \wedge \langle I', S'\rangle \in \mathcal{LC}^t$. Finally, by Def. 24, $(S \overset{\delta_1}{\Longrightarrow}_i S^m \overset{\alpha}{\Longrightarrow}_i S^n \overset{\delta_2}{\Longrightarrow}_i S') \Rightarrow S \overset{\alpha}{\Longrightarrow}_i S'$

      ii. $I^n \overset{\delta_2}{\not\dashrightarrow}_o I'$: S^n is not required to match δ_2 and $\langle I', S'\rangle$ is not required in $\mathcal{LC}^t$.

   (b) $I \overset{\delta_1}{\not\dashrightarrow}_o I^m$: S is not required to match δ_1 and $\langle I^m, S^m\rangle$ is not required in $\mathcal{LC}^t$. □

Lemma 6 *Given that $\mathcal{LC}^t$ is a timed logic conformance:*

$$\forall\, \beta \in \overline{\mathcal{A}} \cup \{\tau\}[(\langle I,S\rangle \in \mathcal{LC}^t \wedge I \overset{\widehat{\beta}}{\Longrightarrow}_i I') \Rightarrow (S \overset{\widehat{\beta}}{\Longrightarrow}_i S' \wedge \langle I',S'\rangle \in \mathcal{LC}^t)]$$

Proof

From Def. 24, first disjunct: $I(\xrightarrow{\tau})^* I^m \xrightarrow{\widehat{\beta}} I^n(\xrightarrow{\tau})^* I'$. Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 25, every I^1, $I \xrightarrow{\tau} I^1$ is matched by some S^1 such that $S \overset{\widehat{\tau}}{\Longrightarrow}_i S^1 \wedge \langle I^1,S^1\rangle \in \mathcal{LC}^t$. This is true inductively for more $\tau \in \tau^*$ and $I^1 \cdots I^m$ and $S^1 \cdots S^m$ according to Formula 25. According to Formula 25, $I^m \xrightarrow{\beta} I^n$ is matched by some S^n such that $S^m \overset{\widehat{\beta}}{\Longrightarrow}_i S^n \wedge \langle I^n,S^n\rangle \in \mathcal{LC}^t$. When $\beta = \tau$, $\widehat{\beta} = 0$, $I^m \xrightarrow{0} I^n$, $m = n$, so, $S^m \overset{0}{\Longrightarrow}_i S^n \wedge \langle I',S'\rangle \in \mathcal{LC}^t$. And finally, by Formula 25 any sequence of τ's in $I^n \xrightarrow{\tau^*} I'$ is matched by $S^n \xrightarrow{\widehat{\tau^*}} S' \wedge \langle I',S'\rangle \in \mathcal{LC}^t$. And, by Def. 24, $(S \overset{\widehat{\tau^*}}{\Longrightarrow}_i S^m \overset{\widehat{\beta}}{\Longrightarrow}_i S^n \overset{\widehat{\tau^*}}{\Longrightarrow}_i S') \Rightarrow S \overset{\widehat{\beta}}{\Longrightarrow}_i S'$. □

Lemma 7 *Given that $\mathcal{LC}^t$ is a timed logic conformance:*

$$\forall\, \delta \in \mathbb{R}[(\langle I,S\rangle \in \mathcal{LC}^t \wedge S \overset{\delta}{\Longrightarrow}_o S' \wedge \exists\, \alpha \in \mathcal{A}[S' \overset{\alpha}{\Longrightarrow}_o]) \Rightarrow (I \overset{\delta}{\Longrightarrow}_o I' \wedge \langle I',S'\rangle \in \mathcal{LC}^t \vee ob(I,\delta,S,\mathcal{LC}^t))]$$

Proof

1. Case $S(\xrightarrow{\tau})^* S^m \xrightarrow{\delta} S^n(\xrightarrow{\tau})^* S'$ (Def. 25, first disjunct):

   Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 23, every S^1, $S \xrightarrow{\tau} S^1$ is matched by some I^1 such that $I \overset{\widehat{\tau}}{\Longrightarrow}_o I^1 \wedge \langle I^1,S^1\rangle \in \mathcal{LC}^t$. This is true inductively for more $\tau \in \tau^*$ and $S^1 \cdots S^m$ and $I^1 \cdots I^m$ according to Formula 23, so $I \overset{\widehat{\tau}}{\Longrightarrow}_o I^m \wedge \langle I^m,S^m\rangle \in \mathcal{LC}^t$. Since $\exists\, \alpha \in \mathcal{A}[S' \overset{\alpha}{\Longrightarrow}_o]$, $S^m \xrightarrow{\delta} S^n \Rightarrow S^m \overset{\delta}{\dashrightarrow}_i S^n$ by Def. 26, and $\exists\, I^n \in S_I$ such that $I^m \overset{\delta}{\Longrightarrow}_o I^n \wedge \langle I^n,S^n\rangle \in \mathcal{LC}^t \vee ob(I,\delta,S,\mathcal{LC}^t)$ according to Formula 26.

(a) $I^m \overset{\delta}{\Longrightarrow}_o$: Then $I^m \overset{\delta}{\Longrightarrow}_o I^n \wedge \langle I^n, S^n \rangle \in \mathcal{LC}^t$ by Formula 26, and by Formula 23 any sequence of τ's in $S^n \xrightarrow{\tau^*} S'$ is matched by $I^n \xrightarrow{\widehat{\tau^*}} I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t$. And, by Def. 25, $(I \overset{\widehat{\tau^*}}{\Longrightarrow}_o I^m \overset{\delta}{\Longrightarrow}_o I^n \overset{\widehat{\tau^*}}{\Longrightarrow}_o I') \Rightarrow I \overset{\delta}{\Longrightarrow}_o I'$

(b) $I^m \overset{\delta}{\not\Longrightarrow}_o$:

i. $\exists_{i \in \{0 \ldots m\}}[I^i \overset{\delta}{\Longrightarrow}_o I']$: Then I^i is another I^m and case $I^m \overset{\delta}{\Longrightarrow}_o$ (above) applies.

ii. $\nexists_{i \in \{0 \ldots m\}}[I^i \overset{\delta}{\Longrightarrow}_o]$: Then $\forall_{i \in \{0 \ldots m\}}[ob(I^i, \delta, S^i, \mathcal{LC}^t)]$, and in particular according to Def. 28 $ob(I, \delta, S, \mathcal{LC}^t)$.

2. Case $S \xrightarrow{\delta_1} S^m(\xrightarrow{\tau})^* S^n \xrightarrow{\delta_2} S' \wedge \delta_1, \delta_2 \in \mathbb{R} \wedge \delta = \delta_1 + \delta_2$ (Def. 25, second disjunct):

(a) $I \overset{\delta_1}{\Longrightarrow}_o$: Since $\exists\ \alpha \in \mathcal{A}[S' \overset{\alpha}{\Longrightarrow}_o]$, $S \xrightarrow{\delta_1} S^m \Rightarrow S \overset{\delta_1}{\dashrightarrow}_i S^n$ by Def. 26. Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 26, $S \overset{\delta_1}{\dashrightarrow}_i S^m$ is matched by some I^m such that $I \overset{\delta_1}{\Longrightarrow}_o I^m \wedge \langle I^m, S^m \rangle \in \mathcal{LC}^t$. Since $\langle I^m, S^m \rangle \in \mathcal{LC}^t$, by Formula 23, any sequence of τ's in $S^m \xrightarrow{\tau^*} S^n$ is matched by $I^m \xrightarrow{\widehat{\tau^*}} I^n \wedge \langle I^n, S^n \rangle \in \mathcal{LC}^t$.

i. $I^n \overset{\delta_2}{\Longrightarrow}_o$: Since $\exists\ \alpha \in \mathcal{A}[S' \overset{\alpha}{\Longrightarrow}_o]$, $S^n \xrightarrow{\delta_2} S' \Rightarrow S^n \overset{\delta_2}{\dashrightarrow}_i S'$ by Def. 26. Since $\mathcal{LC}^t$ is a timed logic conformation and $\langle I^n, S^n \rangle \in \mathcal{LC}^t$, according to Formula 26, $S^n \overset{\delta_2}{\dashrightarrow}_i S'$ is matched by some I' such that $I^n \overset{\delta_2}{\Longrightarrow}_o I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t$. And, $I \overset{\delta_1}{\Longrightarrow}_o I^m \overset{\widehat{\tau^*}}{\Longrightarrow}_o I^n \overset{\delta_2}{\Longrightarrow}_o I' \Rightarrow I \overset{\delta}{\Longrightarrow}_o I'$ by Def. 25.

ii. $I^n \overset{\delta_2}{\not\Longrightarrow}_o$, $(ob(I^n, \delta_2, S^n, \mathcal{LC}^t)$ holds): Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 26, and the Def. 28, $\exists\ I' \in S_I, \delta_2' \in \mathbb{R}[I^n \overset{\delta_2'}{\Longrightarrow}_o I' \wedge S^n \xrightarrow{\delta_2} S' \Rightarrow \langle I', S' \rangle \in \mathcal{LC}^t]$

A. $\exists_{i \in \{m \ldots n\}}[I^i \overset{\delta_2}{\Longrightarrow}_o I']$:Then I^i is another I^n and case $I^n \overset{\delta_2}{\Longrightarrow}_o$ (above) applies.

B. $\not\exists_{i \in \{m \ldots n\}}[I^i \overset{\delta_2}{\Longrightarrow}_o]$: Then $\forall_{i \in \{m \ldots n\}}[ob(I^i, \delta_2, S^i, \mathcal{LC}^t)]$, and in particular according to Def. 28 $ob(I^m, \delta_2, S^m, \mathcal{LC}^t)$. The modeling constraints imposed in Definition 30 on page 63 ensure that all $I \overset{\delta_1}{\rightarrow}$ transitions are derived from a location with a monotonically stronger invariant than I^m, so since $I^m \overset{\delta_2}{\not\Longrightarrow}_o$ there can be no $I \overset{\delta}{\Longrightarrow}_o$ such that $\delta = \delta_1 + \delta_2$. Therefore $I \overset{\delta}{\not\Longrightarrow}_o$ for $\delta = \delta_1 + \delta_2$, and $ob(I, \delta, S, \mathcal{LC}^t)$ holds.

(b) $I \overset{\delta_1}{\not\Longrightarrow}_o$, $(ob(I, \delta_1, S, \mathcal{LC}^t)$ holds): By Def. 28, $\delta > \delta_1 \Rightarrow ob(I, \delta, S, \mathcal{LC}^t)$ □

Lemma 8 *Given that $\mathcal{LC}^t$ is a timed logic conformance:*

$$\forall\, \delta \in \mathbb{R}[(\langle I, S\rangle \in \mathcal{LC}^t \wedge I \overset{\delta}{\Longrightarrow}_i I' \wedge \exists\, \beta \in \overline{\mathcal{A}} \cup \{\tau\}[I' \overset{\beta}{\Longrightarrow}_i]) \Rightarrow (S \overset{\delta}{\Longrightarrow}_i S' \wedge \langle I', S'\rangle \in \mathcal{LC}^t)]$$

Proof

1. Case $I(\xrightarrow{\tau})^* I^m \xrightarrow{\delta} I^n(\xrightarrow{\tau})^* I'$ (Def. 24, first disjunct):

   Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 25, every I^1, $I \xrightarrow{\tau} I^1$ is matched by some S^1 such that $S \overset{\hat{\tau}}{\Longrightarrow}_i S^1 \wedge \langle I^1, S^1\rangle \in \mathcal{LC}^t$. This is true inductively for more $\tau \in \tau^*$ and $I^1 \cdots I^m$ and $S^1 \cdots S^m$ according to Formula 25. Since $\exists\, \beta \in \overline{\mathcal{A}} \cup \{\tau\}[I' \overset{\beta}{\Longrightarrow}_i]$, it is true that $I^m \overset{\delta}{\dashrightarrow}_o I^n$ by Def. 27. According to Formula 27, $I^m \overset{\delta}{\dashrightarrow}_o I^n$ is matched by some S^n such that $S^m \overset{\delta}{\Longrightarrow}_i S^n \wedge \langle I^n, S^n\rangle \in \mathcal{LC}^t$. And finally, by Formula 25 any sequence of τ's in $I^n \xrightarrow{\tau^*} I'$ is matched by $S^n \xrightarrow{\widehat{\tau^*}} S' \wedge \langle I', S'\rangle \in \mathcal{LC}^t$. And, by Def. 24, $(S \overset{\widehat{\tau^*}}{\Longrightarrow}_i S^m \overset{\delta}{\Longrightarrow}_i S^n \overset{\widehat{\tau^*}}{\Longrightarrow}_i S') \Rightarrow S \overset{\delta}{\Longrightarrow}_i S'$.

2. Case $I \xrightarrow{\delta_1} I^0(\xrightarrow{\tau})^* I^n \xrightarrow{\delta_2} I' \wedge \delta_1, \delta_2 \in \mathbb{R} \wedge \delta = \delta_1 + \delta_2$ (Def. 24, second disjunct):

   Since $\exists\, \beta \in \overline{\mathcal{A}} \cup \{\tau\}[I' \overset{\beta}{\Longrightarrow}_i]$, it is true that $I \overset{\delta_1}{\dashrightarrow}_o I^0$ by Def. 27. Since $\mathcal{LC}^t$ is a timed logic conformation, according to Formula 27, every I^0, $I \overset{\delta_1}{\dashrightarrow}_o I^0$ is matched by some S^0 such that $S \overset{\delta_1}{\Longrightarrow}_i S^0 \wedge \langle I^0, S^0\rangle \in \mathcal{LC}^t$. According to Formula 25, every I^1, $I^0 \xrightarrow{\tau} I^1$ is matched by some S^1 such that $S^0 \overset{\hat{\tau}}{\Longrightarrow}_i S^1 \wedge \langle I^1, S^1\rangle \in \mathcal{LC}^t$. This is true inductively for more $\tau \in \tau^*$ and $I^1 \cdots I^n$ and $S^1 \cdots S^n$ according to Formula 25. Since $\exists\, \beta \in \overline{\mathcal{A}} \cup \{\tau\}[I' \overset{\beta}{\Longrightarrow}_i]$, it is true that $I^n \overset{\delta_2}{\dashrightarrow}_o I'$ by Def. 27. According to Formula 27, $I^n \overset{\delta_2}{\dashrightarrow}_o I'$ is matched by some S' such that $S^n \overset{\delta_2}{\Longrightarrow}_i S' \wedge \langle I', S'\rangle \in \mathcal{LC}^t$. And, by Def. 24, $(S \overset{\delta_1}{\Longrightarrow}_i S^n \overset{\widehat{\tau^*}}{\Longrightarrow}_i S^n \overset{\delta_2}{\Longrightarrow}_i S') \Rightarrow S \overset{\delta}{\Longrightarrow}_i S'$. □

Lemma 8 finally establishes that the composition of two timed logic conformances $\mathcal{LC}_1^t\mathcal{LC}_2^t$ is a timed logic conformance.

Lemma 9 *Given a timed logic conformation $\mathcal{LC}^t$, if its inverse $\mathcal{LC}^{t-1}$ is also a timed logic conformation, then $\mathcal{LC}^t$ is a weak timed bisimulation.*

Proof

The proof is structured over the formulas defining $\mathcal{LC}^t$, deriving the formulas defining a weak timed bisimulation when the $\mathcal{LC}^t$ formulas hold in both directions.

1. Formula 22: Conjuncting Formulas 22 and 24 together and reversing the roles of S and I in the transition predicates of Formula 24 yields

$$\begin{aligned} \forall\, S'\ [S \xrightarrow{\alpha} S' \quad &\Rightarrow \quad \exists\, I'\ [I \overset{\alpha}{\Longrightarrow}_o I' \wedge \langle I', S'\rangle \in \mathcal{LC}^t]] \wedge \\ \forall\, S'\ [S \xrightarrow{\alpha} S' \wedge I \overset{\alpha}{\Longrightarrow} \quad &\Rightarrow \quad \exists\, I'\ [I \overset{\alpha}{\Longrightarrow}_i I' \wedge \langle I', S'\rangle \in \mathcal{LC}^t]] \end{aligned}$$

Since $I \overset{\alpha}{\Longrightarrow}_o I'$, and $\alpha \notin \overline{\mathcal{A}}$, the third disjunct of Def. 25 does not apply, so $I \overset{\alpha}{\Longrightarrow}_o I' \Rightarrow I \overset{\alpha}{\Longrightarrow}$ resulting in

$$\begin{aligned} \forall\, S'\ [S \xrightarrow{\alpha} S' \quad &\Rightarrow \quad \exists\, I'\ [I \overset{\alpha}{\Longrightarrow}_o I' \wedge \langle I', S'\rangle \in \mathcal{LC}^t]] \wedge \\ \forall\, S'\ [S \xrightarrow{\alpha} S' \quad &\Rightarrow \quad \exists\, I'\ [I \overset{\alpha}{\Longrightarrow}_i I' \wedge \langle I', S'\rangle \in \mathcal{LC}^t]] \end{aligned}$$

Combining implications results in

$$\forall\, S'\ [S \xrightarrow{\alpha} S' \Rightarrow \exists\, I'\ [I \overset{\alpha}{\Longrightarrow}_o I' \wedge I \overset{\alpha}{\Longrightarrow}_i I' \wedge \langle I', S'\rangle \in \mathcal{LC}^t]]$$

And since $\nexists\ \alpha \in Act[\alpha \in \mathcal{A} \wedge \alpha \in \overline{\mathcal{A}}]$, only the first two disjuncts of both Definitions 24 and 25 are consistent with each other, and they equal the definition of τ-closure (Def. 20),

$$\forall\, S'\ [S \xrightarrow{\alpha} S' \quad \Rightarrow \quad \exists\, I'\ [I \overset{\alpha}{\Longrightarrow} I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]]$$

2. Formula 23: Conjuncting Formulas 23 and 25 and reversing the roles of S and I in Formula 25 transition predicates yields

$$\forall\, S'\ [S \xrightarrow{\beta} S' \quad \Rightarrow \quad \exists\, I'\ [I \overset{\widehat{\beta}}{\Longrightarrow}_o I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]] \wedge$$
$$\forall\, S'\ [S \xrightarrow{\beta} S' \quad \Rightarrow \quad \exists\, I'\ [I \overset{\widehat{\beta}}{\Longrightarrow}_i I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]]$$

which by a subset of the α-case reasoning simplifies to

$$\forall\, S'\ [S \xrightarrow{\beta} S' \quad \Rightarrow \quad \exists\, I'\ [I \overset{\widehat{\beta}}{\Longrightarrow} I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]]$$

3. Formula 26: Conjuncting Formulas 26 and 27 and reversing the roles of S and I in Formula 27 yields

$$\forall\, S'\ [S \overset{\delta}{\dashrightarrow}_i S' \quad \Rightarrow \quad (\exists\, I'\ [I \overset{\delta}{\Longrightarrow}_o I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t] \vee ob(I, \delta, S, \mathcal{LC}^t))] \wedge$$
$$\forall\, S'\ [S \overset{\delta}{\dashrightarrow}_o S' \quad \Rightarrow \quad \exists\, I'\ [I \overset{\delta}{\Longrightarrow}_i I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]]$$

Since $\delta \in \mathbb{R}$ transitions are added to all three extended transition relations in the same way $\overset{\delta}{\Longrightarrow} \triangleq \overset{\delta}{\Longrightarrow}_i \triangleq \overset{\delta}{\Longrightarrow}_o$ the consequents of the implications can be changed as follows:

$$\forall\, S'\ [S \overset{\delta}{\dashrightarrow}_i S' \quad \Rightarrow \quad (\exists\, I'\ [I \overset{\delta}{\Longrightarrow} I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t] \vee ob(I, \delta, S, \mathcal{LC}^t))] \wedge$$
$$\forall\, S'\ [S \overset{\delta}{\dashrightarrow}_o S' \quad \Rightarrow \quad \exists\, I'\ [I \overset{\delta}{\Longrightarrow} I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]]$$

In order to unite the antecedents, Table 4 reports all possible combinations of truth values for the three clauses under the assumptions that $\mathcal{LC}^t$ and $\mathcal{LC}^{t-1}$ are both logic conformations and that $\langle I, S\rangle \in \mathcal{LC}^t \wedge S \xrightarrow{\delta} S'$

Table 4. Delta Predicate Truth Table.

$S \overset{\delta}{\dashrightarrow}_o S'$	$S \overset{\delta}{\dashrightarrow}_i S'$	$ob(I, \delta, S, \mathcal{LC}^t)$	Conclusion
0	0	0	See case 000.
0	0	1	*See case 0X1.
0	1	0	$I \overset{\delta}{\Longrightarrow} I' \wedge \langle I', S'\rangle \in \mathcal{LC}^t$
0	1	1	*See case 0X1
1	0	0	$I \overset{\delta}{\Longrightarrow} I' \wedge \langle I', S'\rangle \in \mathcal{LC}^t$
1	0	1	*See case 1X1.
1	1	0	$I \overset{\delta}{\Longrightarrow} I' \wedge \langle I', S'\rangle \in \mathcal{LC}^t$
1	1	1	*See case 1X1.
* Contradiction, impossible case.			

(a) **000:** Since $S \overset{\delta}{\not\dashrightarrow}_o S'$ and $S \overset{\delta}{\not\dashrightarrow}_i S'$ there are no inputs, outputs, or τ's possible from S' or its derivatives. Therefore there is no upper bound on S or S' δ-transitions because modeling constraints restrict upper bounds to output- or τ-capable locations. Since only non-Zeno automatons are allowed, time must continue to progress forever. Further, future I' states must exhibit the same behavior, for if there were any future non-delta actions possible from some I' they would have to be possible from the corresponding S'. This includes inputs, because Formula 22 has to hold over $\mathcal{LC}^{t-1}$, and it also includes τ's because the modeling constraints would upper-bound time progression from I and $ob(I, \delta, S, \mathcal{LC}^t)$ would have to be true, and $ob(I, \delta, S, \mathcal{LC}^t)$ is not true. Therefore $S \xrightarrow{\delta} S' \Rightarrow I \overset{\delta}{\Longrightarrow} I' \wedge \langle I', S'\rangle \in \mathcal{LC}^t$.

(b) **0X1:** $ob(I, \delta, S, \mathcal{LC}^t) \Rightarrow I \overset{\delta_1}{\Longrightarrow}_o I' \xrightarrow{\beta} I'' \wedge \langle I', S'\rangle \in \mathcal{LC}^t$. Since S' must conform to I', $S' \overset{\widehat{\beta}}{\Longrightarrow}$, and $S \xrightarrow{\delta} S' \overset{\widehat{\beta}}{\Longrightarrow} \Rightarrow S \overset{\delta}{\dashrightarrow}_o S'$, a contradiction, so case 0X1 is impossible.

(c) **1X1:** $S \overset{\delta}{\dashrightarrow}_o S' \Rightarrow I \overset{\delta}{\Longrightarrow} I' \Rightarrow \neg ob(I, \delta, S, \mathcal{LC}^t)$, a contradiction, so case 1X1 is impossible.

Since all of the possible cases conclude $I \overset{\delta}{\Longrightarrow} I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t$ the two implications can be united and the antecedent can be generalized to all δ-transitions resulting in

$$\forall\, S'\ [S \xrightarrow{\delta} S' \quad \Rightarrow \quad \exists\, I'\ [I \overset{\delta}{\Longrightarrow} I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]]$$

Since $\alpha \in \mathcal{A}, \beta \in \overline{\mathcal{A}} \cup \{\tau\}$, and $\delta \in \mathbb{R}$ cover the domain of γ in Def. 21, and $\forall \alpha \in \mathcal{A}\ [\widehat{\alpha} = \alpha]$ and $\forall \delta \in \mathbb{R}\ [\widehat{\delta} = \delta]$, the three results from above can be combined into the single formula:

$$\forall\, S'\ [S \xrightarrow{\gamma} S' \quad \Rightarrow \quad \exists\, I'\ [I \overset{\widehat{\gamma}}{\Longrightarrow} I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]]$$

Further, since the same pairs of formulas hold for all α-, β-, and δ-transitions that I can do

$$\forall\, I'\ [I \xrightarrow{\gamma} I' \quad \Rightarrow \quad \exists\, S'\ [S \overset{\widehat{\gamma}}{\Longrightarrow} S' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]]$$

therefore $\mathcal{LC}^t$ is a weak timed bisimulation:

$$\begin{aligned} &\forall\, \langle I, S \rangle \in \mathcal{LC}^t, \gamma \in \mathit{Act} \cup \mathbb{R}[\\ &\qquad \forall\, S'\ [S \xrightarrow{\gamma} S' \quad \Rightarrow \quad \exists\, I'\ [I \overset{\widehat{\gamma}}{\Longrightarrow} I' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]] \wedge \\ &\qquad \forall\, I'\ [I \xrightarrow{\gamma} I' \quad \Rightarrow \quad \exists\, S'\ [S \overset{\widehat{\gamma}}{\Longrightarrow} S' \wedge \langle I', S' \rangle \in \mathcal{LC}^t]]] \quad \square \end{aligned}$$

The final property that must be established about timed logic conformations is that the union of a set of timed logic conformations is a timed logic conformation.

Lemma 10

$$\bigcup \mathcal{LC}_i^t \tag{32}$$

is a timed logic conformation.

Proof

For every pair of states $\langle P, R\rangle \in \bigcup \mathcal{LC}_i^t$ and for every possible action the Formulas 22 through 27 hold for some $\mathcal{LC}_i^t$; so, by Def. 29, and the definition of union, they hold for $\bigcup \mathcal{LC}_i^t$; therefore $\bigcup \mathcal{LC}_i^t$ is a timed logic conformation. □

4.8 Timed Logic Conformance as a Maximum Fixpoint

The definition of $\mathcal{LC}^t$ must be narrowed down so that it uniquely defines one of the many possible relations between DLTS automata states. There are many possible solutions to the relation $\mathcal{LC}^t$ as defined, including $\mathcal{LC}^t = \varnothing$. The one $\mathcal{LC}^t$ relation of particular interest is the largest relation known as the **maximum fixpoint** of $\mathcal{LC}^t$. $\mathcal{LC}^t$'s maximum fixpoint is useful because implementation DLTS I can be safely substituted for a specification DLTS S when the initial states of I and S are in the maximum fixpoint $\mathcal{LC}^t$ relation. The following definitions and claims are made for DLTS induced from TSA conforming to the modeling constraints enumerated in Definition 30.

Definition 32. Timed Logic Conformation Maximum Fixpoint: $\widehat{\mathcal{LC}^t}$.

$$\widehat{\mathcal{LC}^t} \triangleq \bigcup_{\mathcal{R} \in \wp(S_I \times S_S)} \{\mathcal{R} \mid \mathcal{R} \textit{ is a timed logic conformation}\}$$

Theorem 3 *Given Def. 32, $\widehat{\mathcal{LC}^t}$ is the largest timed logic conformation.*

Proof

By Lemma 10, $\widehat{\mathcal{LC}^t}$ is a timed logic conformation, and by definition it includes any other such. □

Definition 33. Timed Logic Conformant DLTS: $I\ {}_{o}\underline{\diamond}_{i}\ S$. *Two DLTS's* $I \triangleq \langle S_I, Act_I, \longrightarrow_I, \langle l_0, \pi_0\rangle_I\rangle$, *and* $S \triangleq \langle S_S, Act_S, \longrightarrow_S, \langle l_0, \pi_0\rangle_S\rangle$, *induced from TSA conforming to the Definition 30* *modeling constraints are timed logic conformant (written $I \,{}_o\!\overset{\diamond}{\preceq}_i\, S$) iff*

$$\langle\langle l_0, \pi_0\rangle_I, \langle l_0, \pi_0\rangle_S\rangle \in \widehat{\mathcal{LC}^t} \tag{33}$$

Theorem 4 *Given Def. 32, ${}_o\!\overset{\diamond}{\preceq}_i$ is a partial order.*

Proof

1. Reflexive: For any DLTS P, $P \,{}_o\!\overset{\diamond}{\preceq}_i\, P$ by Lemma 2 since $\langle\langle l_0, \pi_0\rangle_P, \langle l_0, \pi_0\rangle_P\rangle \in \widehat{\mathcal{LC}^t}$.

2. Transitive: For three DLTS P, Q, and R, $P \,{}_o\!\overset{\diamond}{\preceq}_i\, Q \wedge Q \,{}_o\!\overset{\diamond}{\preceq}_i\, R \Rightarrow P \,{}_o\!\overset{\diamond}{\preceq}_i\, R$ by Lemma 3 since $\langle\langle l_0, \pi_0\rangle_P, \langle l_0, \pi_0\rangle_Q\rangle \in \widehat{\mathcal{LC}^t_1} \wedge \langle\langle l_0, \pi_0\rangle_Q, \langle l_0, \pi_0\rangle_R\rangle \in \widehat{\mathcal{LC}^t_2} \Rightarrow \langle\langle l_0, \pi_0\rangle_P, \langle l_0, \pi_0\rangle_R\rangle \in \widehat{\mathcal{LC}^t_1\mathcal{LC}^t_2}$.

3. Antisymmetric: For two DLTS P and Q, $P \,{}_o\!\overset{\diamond}{\preceq}_i\, Q \wedge Q \,{}_o\!\overset{\diamond}{\preceq}_i\, P \Rightarrow P \approx Q$ per Lemma 9 since $(\langle\langle l_0, \pi_0\rangle_P, \langle l_0, \pi_0\rangle_Q\rangle \in \widehat{\mathcal{LC}^t} \wedge \langle\langle l_0, \pi_0\rangle_Q, \langle l_0, \pi_0\rangle_P\rangle \in \widehat{\mathcal{LC}^t}^{-1}) \Rightarrow \langle\langle l_0, \pi_0\rangle_Q, \langle l_0, \pi_0\rangle_P\rangle \in \widehat{\mathcal{W}}$. □

Finally, ${}_o\!\overset{\diamond}{\preceq}_i$ is overloaded to relate TSA conforming to the modeling constraints enumerated in Definition 30.

Definition 34. Timed Logic Conformant TSA: *$I \,{}_o\!\overset{\diamond}{\preceq}_i\, S$. An implementation TSA I is* **timed logic conformant** *to specification TSA S (written $I \,{}_o\!\overset{\diamond}{\preceq}_i\, S$) iff both I and S conform to the Definition 30 modeling constraints and the DLTS induced from I, $I' = \langle S_I, Act_I, \longrightarrow_I, \langle l_0, \vec{0}\rangle_I\rangle$, and the DLTS induced from S, $S' = \langle S_S, Act_S, \longrightarrow_S, \langle l_0, \vec{0}\rangle_S\rangle$, are timed logic conformant (i.e., $I' \,{}_o\!\overset{\diamond}{\preceq}_i\, S'$).*

4.9 TLC, Parallel Composition, and Hierarchical Verification

Historically, one of the most theoretically important properties of equivalence and partial order relations between models of concurrent systems is whether or not they are preserved by parallel composition operations. If a relation is preserved by the composition process, and the input assumptions of each of the components in the composition are independently specified, then the verification task can be broken down into independent pieces without resorting to assumes-guarantees-style proof obligations.

If parallel composition does not preserve the relation then designers must verify every composition against a higher level of abstraction, and they should eventually have a top-level specification that is not parallel composed. Often this is not practical for real-world designs because of the complexity of generating a monolithic top-level specification, and in that case one should carefully simulate the top-level composition to ensure it behaves as expected. One should also employ model checkers to prove it has the important properties they expect and that the composed specification is free from deadlock and livelock.

When the input assumptions of each of the components are not independently specified, then designers must do the extra assumes-guarantees-style verifications to determine whether or not the implicit input assumptions of the components operating together and in an environment are satisfied together.

Since the TSA formalism supports specifying the input assumptions of each component, and the CI-free property of compositions ensures that the input assumptions of cooperating components do not interfere with outputs, an efficient top-down verification methodology can be realized. Top-down hierarchical TLC verification starts at the most abstract level with a specification that incorporates the environmental timing issues (e.g., input frequency, stimulus-response constraints) into its behavior. The specification is the contract with the environment; as such it defines the behavior required for the inputs it accepts. Only implementations that satisfy the TLC relation with the specification fulfill the contract; TLC failures are design errors.

Then, a hierarchical system is top-down verified by defining (by parallel composition) a set of sub-specifications that are TLC-verified against the specification. Sub-specifications must also be CI-free, but only in the reachable subset of their state space explored by the TLC-relation with the specification's reachable state space. Designers continue down the hierarchy, TLC-verifying each sub-specification against its sub-sub-specification until TLC holds with implementations composed entirely of design primitives. The reverse method can be used from the bottom-up to create systems (as done in the STARI example 6.2.3).

Unfortunately, as currently defined, TLC is not preserved by parallel composition. Figure 7 illustrates the problem. I and S are nearly identical; the only difference is the guard $ki \geq 1$ on $I1 \xrightarrow{a_-} I2$. Formula 23 forgives this difference when verifying $I \,{}_o\underline{\diamond}_i\, S$, since $\forall\, ks \in [0,1)[S1_{ks} \xrightarrow{a_-} S2_{ks} \Rightarrow I1_{ks} \overset{1-ks}{\longrightarrow} I1_1 \xrightarrow{a_-} I2_1 \Rightarrow I1_{ks} \overset{a_-}{\Longrightarrow}_i I2_1 \wedge \langle I2_1, S2_{ks}\rangle \in \mathcal{LC}^t]$. The difference is forgiven because $I1$'s a_- outputs are allowed to be a timed subset of $S1$'s a_- outputs. Allowing the implementation to match the specification using $\Longrightarrow_o$ keeps $\langle I1_k, S1_k\rangle$ in $\mathcal{LC}^t$ for $k \in [0,1)$.

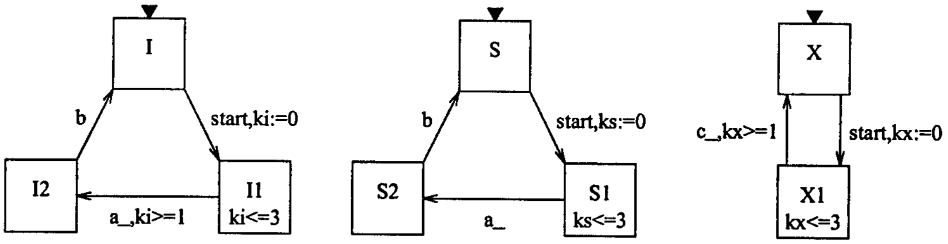


Figure 7. $I \,{}_o\underline{\diamond}_i\, S \not\Rightarrow I \| X \,{}_o\underline{\diamond}_i\, S \| X$.

However, in composition with $X1$, states $\{\langle (I2\|X1)_1, (S2\|X1)_k\rangle \mid k \in [0,1)\}$ cannot be in $\mathcal{LC}^t$ because Formula 25 (checking that the specification can match implementation outputs) fails when $(I2\|X1)_1 \xrightarrow{c_-}$ and $(S2\|X1)_k \overset{c_-}{\not\Longrightarrow}_i$. State $(S2\|X1)_{kx}$ cannot do $X1$'s c_- for $kx \in [0,1)$ because the guard $kx \geq 1$ is false. This means that state pairs $\langle (I1\|X1)_k, (S1\|X1)_k\rangle$ for $k \in [0,1)$ cannot be in $\mathcal{LC}^t$, and neither can states $\langle (I\|X)_k, (S\|X)_k\rangle$. Therefore $I\|X \,{}_o\underline{\diamond}_i\, S\|X$ does not hold.

In this example, allowing time to progress in $I1$ to match $S1$'s a_- output and the unmatched changes in $X1$'s behavior causes the problem. $X1$ behaves differently when $kx \geq 1$ than it does when $kx < 1$. The TLC relation cannot allow this difference in the composition even though it accepted the difference between $I1$ and $S1$ via $I1 \Longrightarrow_o$ while computing $I \,{}_o\underline{\diamond}_i\, S$.

This example illustrates the conflict between parallel composition and relaxing the timing relationship between implementation and specification outputs. The TSA time successor Rule 7 on page 41 defines that time progresses equally for all DLTS's induced from parallel TSA, but neither the TLC relation, nor the modeling constraints, nor the CI-free property prohibit transition guards from becoming enabled (like $X1 \xrightarrow{c-,kx\geq 1} X$'s) while time progresses.

This is not a problem for CTR verification because α in Formula 2's antecedent $S \xrightarrow{\alpha} S'$ does not range over $\{i\}$ for the time prefix $[0,3]$. This means that $[0,3].(S1||X1) \overset{i}{\not\longrightarrow}$, so $(S1||X1) \xrightarrow{a-}$ is not enabled. Only when I's delay prefix becomes $[0,2].(I1||X1) \xrightarrow{a-}$ does Formula 1 on page 25 require them to match, and then they do, because $[0,2].X1 \xrightarrow{c-}$ is enabled for both compositions at that point. If Formula 23 were changed to match CTR's semantics from

$$\forall\, S'\ [S \xrightarrow{\beta} S' \Rightarrow \exists\, I'\ [I \overset{\widehat{\beta}}{\Longrightarrow}_o I' \wedge \langle I', S'\rangle \in \mathcal{LC}^t]]$$

to

$$S \xrightarrow{\beta}\ \ \Rightarrow\ \ \exists\, \delta, I', I'', S', S''\ [I \overset{\delta}{\Longrightarrow} I' \overset{\beta}{\Longrightarrow} I'' \wedge S \overset{\delta}{\Longrightarrow} S' \xrightarrow{\beta} S'' \wedge$$
$$\{\langle I', S'\rangle, \langle I'', S''\rangle\} \subseteq \mathcal{LC}^t]$$

this formulation would not require the original state S^* reached via $S \xrightarrow{\beta} S^*$ to be matched. It relaxes TLC such that the example in Figure 7 would preserve the composition. Unfortunately, if there were also an input transition $X1 \xrightarrow{d,kx\leq 1} X1$, TLC would hold, despite the fact that $(S||X)$ leads to an input that $(I||X)$ cannot match. Such an error is not consistent with the notion of an acceptable implementation; so TLC cannot be weakened like CTR.

In contrast, if the TLC relation is strengthened such that the implementation must match output timing exactly (i.e., replace $\Longrightarrow_o$ by $\Longrightarrow$ in all Formulas 22 through 27), then composition preserves TLC, but it makes the relation impractical. When implementations are required to match specification output timing exactly, TLC generally rejects useful abstractions because typical implementations do not always exhibit both the best-case and worst-case delays from all states.

Hence the TLC relation cannot generally be preserved in a practical way by strengthening output timing matching.

The fact that TLC is not preserved by parallel composition is a practical problem only when all compositions are not verified. All compositions are verified except when a monolithic top-level specification cannot be constructed to verify the most abstract composition. In this case, one should simulate and model-check the most abstract composition as described in Section 4.9, and TLC verify it against the composition of the entire next level of abstraction TSA. For example, referring to the TSA modeling hierarchy in Figure 8, in order to verify the top-level composition $A||B||C$, verification of

$$A1||A2||A3||B1||B2||B3||C1||C2||C3 \,{}_{o}\underline{\diamond}_{i}\, A||B||C$$

is required instead of relying on

$$(A1||A2||A3 \,{}_{o}\underline{\diamond}_{i}\, A) \wedge (B1||B2||B3 \,{}_{o}\underline{\diamond}_{i}\, B) \wedge (C1||C2||C3 \,{}_{o}\underline{\diamond}_{i}\, C)$$

to separately satisfy the intended behavior of $A||B||C$. Any TLC failures in the large verification should be carefully scrutinized to ensure the specification is requiring the appropriate behavior. If not, then modify A, B, or C; if so, correct the offending A_i, B_i, or C_i.

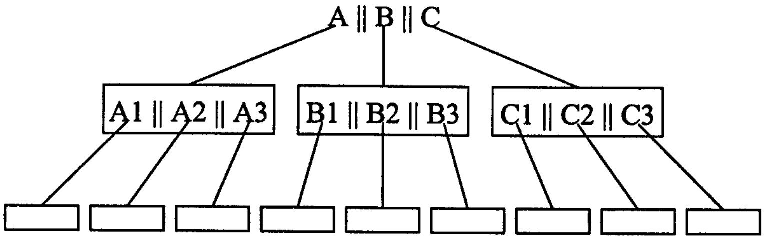


Figure 8. Parallel Specification Hierarchy.

Although this may seem just as expensive as the assumes-guarantees reasoning process, note that all of the models in this example are models of the system being built, not of the environment.

In Figure 3, the problem depicted is the verification of a system being built—perhaps Y, where X and Z are models of the environment interacting with Y. Here, the environmental constraints are captured in the TSA $A||B||C$ which is at the same level of abstraction as Y. The verifications depicted in Figure 8 represent two levels of abstraction below Y, and reasoning about the system itself, not the environment. If Y must be concretized to be implemented, then here are even more assumes-guarantees verifications necessary for all the Y_i timed processes in their environments.

4.10 Summary

This chapter formally defined the timed equivalence relation weak timed bisimulation that relates DLTSs with different internal action sequences but the same observable action sequences and timing. To relate systems that do not have the exact same timing, it defined how to abstract away the temporal differences between TSA, and how to use those abstractions to weaken weak timed bisimulation via the partial order Timed Logic Conformance.

With a few well-defined exceptions, Timed Logic Conformance requires that implementation inputs are a timed superset of specification inputs and that implementation outputs are a timed subset of specification outputs. Timed Logic Conformance formalizes these notions and specifies when an implementation can safely replace a specification, and it has the necessary mathematical properties to support hierarchical verification of large systems with the exception that one must be careful when the most abstract specification is parallel composed.

The TLC verification process supports a powerful and efficient top-down verification methodology that also works bottom up. The TLC verification methodology is better than assumes-guarantees reasoning because it simplifies and reduces the burden of building models, and it breaks the verification down into less complex independent pieces.

TLC verification simplifies model building because fewer models have to be built; no models of the environment itself need to be constructed, and no models of the rest of the system and the environment (the environment from a particular component's point of view) need to be constructed. Further, the system models that are constructed are simpler because not all inputs in all states for all times have to be defined. Only the inputs necessary to satisfy the CI-free property have to be added.

The TLC verification methodology is simpler because it can be independently decomposed without the assumes-guarantee circular dependency verifications. This reduces the magnitude of the verification task tremendously because iteratively changing models and specifications only affect the verifications up and down the hierarchy, not across the breadth of it for every iteration.

V. Timed Logic Conformance System

This chapter describes the Prolog Timed Logic Conformance System (TLCS). After presenting some background information, it describes the finite automata induced from Timed Safety Automata (TSA) called region automata. Region automata are useful for reasoning about TSA behavior using computers, and in particular for this research using TLCS. After describing the TLCS region automata time representation, the chapter describes the TLCS rules and procedures that efficiently implement the TLC decision procedure. Finally, it concludes with a description of the TLCS TSA input format, TLCS TSA parallel composition, and TLCS user interface.

5.1 Background

TLCS is actually a second generation Prolog program. The first generation program computes the TLC maximum fixpoint over a discrete projection of the induced DLTS. It directly computes the $\Longrightarrow_i$, $\Longrightarrow_o$, $\dashrightarrow_i$, and $\dashrightarrow_o$ transition relations, takes the cross product of the discrete subsets of implementation and specification DLTS states, and whittles the cross product down to the maximum fixpoint $\widehat{\mathcal{LC}^t}$. TLCS uses predicates implementing the TLC definition (Formulas 22 through 27) to reject state-pairs that do not satisfy the relation. The first generation program is quite useful for understanding the subtleties of the TLC definitions, but it is only practical for TSA with handfuls of locations.

The second-generation program (TLCS) verifies whether or not the dense time behavior of two TSA satisfy the TLC relation properties. Instead of verifying the cross product of the state spaces, TLCS examines only the subset of the TLC maximum fixpoint relation that is reachable between the two TSA being compared when they start in their initial locations, time progresses the same for both of them, and they receive the same inputs. TLCS explicitly enumerates the reachable states of the two region automata being compared using a common frame of reference for time passing. TLCS renames the clocks of the two systems to ensure they are unique and unions the renamed clock sets into a single clock set used as a common time reference. TLCS does not directly compute the $\Longrightarrow_i$, $\Longrightarrow_o$, $\dashrightarrow_i$, and $\dashrightarrow_o$ transition relations. Rather, it follows τ and time-passing transitions (δ's) when necessary and allowed by the TLC definition (Def. 29) formulae to determine if TLC holds. For example, if $S \xrightarrow{a-} S'$, but $I \not\xrightarrow{a-}$ TLCS follows $\xrightarrow{\tau}$ or $\xrightarrow{\delta}$ transitions in accordance with the $\Longrightarrow_o$ relation definition (Def. 25) to determine if $I \overset{a}{\Longrightarrow}_o I'$ and if I' and S' satisfy the TLC relation properties.

TLCS depth-first explores the mutually reachable state space of the two automata by taking transitions and advancing time from their initial states to determine if any TLC formula is violated in any reachable state pair. If no TLC formula is violated and all of the reachable state space is explored, TLCS succeeds and TLC holds. If a TLC formula is violated before examining all of the reachable states, TLC fails, and TLCS can be queried to report a **trace** (sequence of actions) or **simulation** (sequence of locations, times, and actions) leading to the failure. TLCS examines all reachable states for all possible actions starting from the initial state pair, so it will detect any failure that disqualifies the initial state pair $\langle I_0, S_0 \rangle$ from being in $\widehat{\mathcal{LC}^t}$. This is true because $\langle I_0, S_0 \rangle \notin \widehat{\mathcal{LC}^t}$ implies for some action $\sigma \in \mathit{Act} \cup \mathbb{R}$ either $I_0 \xrightarrow{\sigma} I$ and $S_0 \xrightarrow{\sigma} S$ and $\langle I, S \rangle \notin \widehat{\mathcal{LC}^t}$ or one of the systems could do σ and the other could not match σ according to the TLC formulae. Both cases are detected by TLCS.

The following sections explain the TLCS implementation. First the region automata time representation is explained, then TLCS data structures and the Prolog rules and procedures reveal the TLCS algorithm that decides if TLC holds between a pair of implementation and specification TSA.

5.2 Region Automata

Since the state space of a DLTS is uncountable, the DLTS semantic model cannot be used to represent TSA and compute relationships between them on finite computer systems. Therefore a finite representation of the DLTS called region automata from Alur and Dill (ACD90) and fully developed in (AD94) is adapted for TSA semantics and computing TLC.

The main difference between DLTS and region automata is that the uncountable number of clock assignments representing the different possible combinations of clock values in the DLTS are represented finitely by a collection of open and closed intervals in the region automata, one interval for each clock, and a relation on clocks that orders them according to the magnitude of the fractional part of the clock value. Hence, the "state" of a region automata consists of a label representing the TSA location, a collection of time intervals, and the fractional-part relation. The intervals and the fractional-part together define equivalence classes for the clock assignments. Figure 9 serves as an example. In this example there are two clocks, x and y. The largest integers used to constrain clocks x and y are 2 and 1 respectively. While there are an uncountable number of real values these two clocks can take on with respect to one another, only 28 different equivalence classes are required to finitely represent the clock assignments as depicted in the figure. Hence, instead of an uncountable number of $\langle l, \vec{\pi} \rangle$ timed states, and transitions between them, a finite number of $\langle l, EC(\vec{\pi}) \rangle$ tuples and transitions represent TSA behavior in the computer (where $EC(\vec{\pi})$ stands for the Equivalence Class of the clock assignment $\vec{\pi}$).

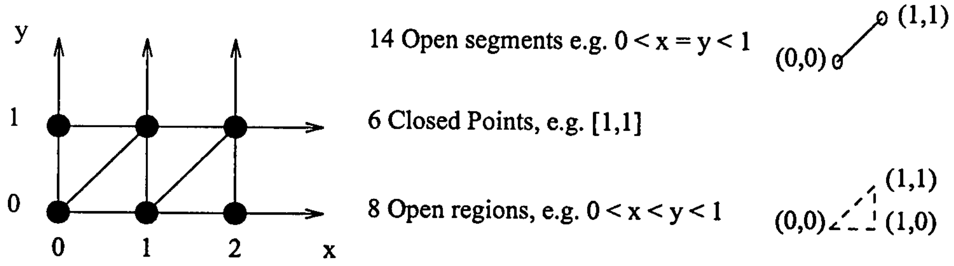

y
1
0
0
1
2
x
14 Open segments e.g. 0 < x = y < 1
(0,0)
(1,1)
6 Closed Points, e.g. [1,1]
8 Open regions, e.g. 0 < x < y < 1
(1,1)
(0,0)
(1,0)

Figure 9. Region Automata Time Regions.

Since only integers are used to constrain TSA, as time progresses the truth of guards and invariants can only change when a clock value changes from an integral to a real value or vice versa. Consequently, when no clocks are integral, the algorithm need only keep track of which two integers each clock is between and which clock(s) will reach their next integral value first (i.e., which clock(s) has(have) the largest fractional part). When one or more clocks are integral, for time to progress, their values will change next, and after the time changes, those clocks have the smallest fractional part, and no clocks are integral. Note that once a clock value exceeds the largest integer used to constrain it, the truth or falsity of guards and invariants referencing it do not change, so there is no longer any need to reference the fractional parts of such clocks. Since time progresses at the same rate in TSA clocks, it can only progress along trajectories parallel to the line x = y in the 2-clock example in Figure 9; e.g., time progresses transitively from point [x = y = 0] to the line segment [0 < x = y < 1] to the point [x = y = 1] to the open region [1 < x < 2, 1 < y < 2] to the line segment [x = 2, y > 1] to the open region [x > 2, y > 1].

TLCS time equivalence classes are called **time regions**. A time region is a tuple `[CV,CC]`, where `CV` (Clock Values) is a sorted sequence of tuples `[[CName,L,R],...]`. `CName` is the clock name, `L` is the lower integer bound on the clock `CName`'s value, and `R` is either the upper integer bound on clock `CName`'s value or the atom '`i`' representing infinity when `CName`'s value is unbounded. Only two kinds of intervals, closed and open are necessary; there is no representation for clopen intervals. An interval is closed and represents a single point when `L`=`R`. An interval is open when `L`≠`R`. `CC` (Clock Classes) is a sequence of tuples containing alphabetically sorted lists of clock names representing a descending-order sorted partition of bounded clock fractional values (e.g., `[[c1,c3],[c2]]` where `c1` and `c3` have equal fractional parts, and `c2` has a smaller fractional part than `c1` and `c3`). Every `CName` in `CV` except those whose value is not bound will be in one and only one element of `CC`. If no clocks are bound, (i.e., all clock values exceed the maximum values constraining them) `CC == []`. TLCS time regions for the time progression example in the previous paragraph are shown in Table 5.

Since neither x or y are reset and time progresses equally on both clocks from $(x, y) = (0, 0)$ in the Table 5 example, the table does not include an example representation for a time region where `frac(x)`≠`frac(y)`. TLCS uses `[[[x,1,2],[y,0,1]],[[x],[y]]]` to represent the time region [1

Table 5. TLCS Time Region Representation.

Region	Representation
$[x = y = 0]$	`[[[x,0,0],[y,0,0]],[[x,y]]]`
$[0 < x = y < 1]$	`[[[x,0,1],[y,0,1]],[[x,y]]]`
$[x = y = 1]$	`[[[x,1,1],[y,1,1]],[[x,y]]]`
$[1 < x < 2, 1 < y < 2]$	`[[[x,1,2],[y,1,i]],[[x]]]`
$[x = 2, y > 1]$	`[[[x,2,2],[y,1,i]],[[x]]]`
$[x > 2, y > 1]$	`[[[x,2,i],[y,1,i]],[]]`

$< x < 2, 0 < y < 1]$ where `frac(x)>frac(y)`. This region is the triangle interior defined by the points $\{(1,0),(2,0),(2,1)\}$.

In order to ensure that clocks progress at the same rate, equivalence classes for clock values that have not yet exceeded their maximum constraint are no coarser than open intervals between two adjacent integral numbers. For example, the clock value $\mathtt{x} = 1.25$ is represented by the open interval `[x,1,2]` in its equivalence class. Since, in general, clocks may be independently reset and therefore will not always be in the same equivalence class, all combinations of equivalence classes up to and including the class representing when their value exceeds the maximum integer used to constrain them are possible. This means that the number of regions grows exponentially with the largest clock constraint value. Given that C is the clockset, and c_x is the largest integer constraining clock $c \in C$, the number of clock regions is bounded by $[| C |! \cdot 2^{|C|} \cdot \Pi_{c_x \in C}(2c_x + 2)]$ (AD94:p203).

Unfortunately, this fine equivalence class granularity is generally necessary to model time progressing uniformly on all clocks and to eliminate region automata behaviors that would not be possible in the corresponding DLTS. The following example illustrates why we need such a fine granularity of equivalence classes. If instead of limiting the largest equivalence class to be less than one time unit wide, the granularity of the region automata equivalence classes is increased to the open intervals between the integers used to constrain the TSA, then some region automata models exist that exhibit behaviors inconsistent with the induced DLTS semantics.

These inconsistencies arise when time is not constrained to pass at the same rate on all system clocks. This is illustrated by the following example. In the Figure 10 TSA/DLTS, no c_ output can occur before b_ is observed. This is the case because the guard on the c_ transition requires $cb - ca > 1$, and ca and cb cannot ever be more than 1 time unit apart unless the a transition from X1 to X2 is taken. In the portion of the region automata state space and the example transitions illustrated in Figure 11 the region where c_ can occur can be reached by following the a transition from X1 to X2 and returning to X1 from X2 via the b_ transition. Unfortunately, the c_-capable region can also be reached by following δ from $(0,0)$ to $(1,1)$, a from $(1,1)$ to $(0,1)$, δ from $(0,1)$ to region $[0 < ca < 1, 1 < cb < 2]$ where another δ that does not necessarily keep time progressing on both clocks equally can take the automata to region $[0 < ca < 1, 2 < cb < \infty]$ where a c_ output can occur even though X2 was never entered. This example illustrates why clock value equivalence classes for clocks that have not yet exceeded their maximum constraint are no coarser than open intervals between two adjacent integral numbers.

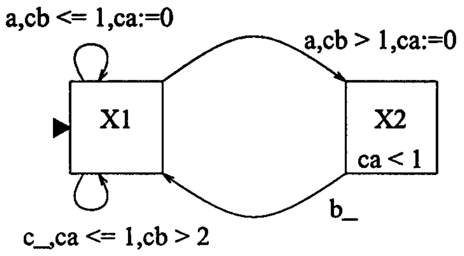


Figure 10. Fine-Grained TSA/DLTS.

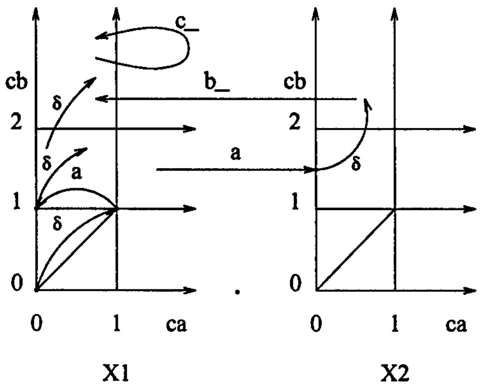


Figure 11. Inconsistent Region Automata with Skewed δ-Transitions.

The time region is a fundamental data structure of the TLCS system. It is an important part of understanding how the TLCS system finitely computes the TLC relationship for dense time. The next important concept is the TLC decision procedure itself.

5.3 TLC Decision Procedure

This section describes and illustrates TLCS's novel and efficient region-automata-based TLC decision procedure implementation. Starting from some basic definitions, the section uses important TLCS code fragments to explain the decision procedure implementation. After the basic definitions, the discussion is broken down into three subsections. The first subsection describes the TLC Prolog query that checks the behavioral part of TLC; the second describes checking a particular formula from the TLC definition, and the third describes how the TLC prolog query verifies time-derivatives.

TLCS inputs are Prolog atoms other than `t` that do not end in an underscore. Outputs are Prolog atoms that end with an underscore (_). Taus are the Prolog atom `t` or Prolog terms `t`(X) where X is a variable representing any Prolog atom. The TLCS queries `input(X)`, `output(X)`, and `tau(X)` are true when `X` is an input, output, or tau action respectively.

In TLCS, a **timed state** of the implementation, specification, and their combined time representation is an `[I,S,T]` tuple where `I` is a Prolog term representing the implementation location, `S` is a Prolog term representing the specification location, and `T` is the time region. The initial timed state is then the initial location of the implementation, the initial location of the specification, and the zero time region of the combined clock set. Since the initial locations of the two TSA being compared must be in the TLC relation between their induced DLTS's and their corresponding region automata at time zero, TLCS starts checking whether or not the two automata satisfy the TLC relation properties from the initial timed state.

TLCS keeps track of all the timed states it has or is currently checking, and once a timed state is checked, it is not checked again. This avoids recomputing tlc/6 for timed states that have already satisfied TLC and for those which are currently being examined in the depth-first trail. In most cases TSA used in this research have recursive behavior patterns, so maintaining and checking the list of visited timed states prevents nonterminating computation.

5.3.1 Behaviorally Checking TLC. The tlc(I,S,Tn,Matched,UnMatched,Parent) Prolog query (abbreviated by the **functor** tlc/6)[1] investigates whether implementation location I and specification S satisfy the TLC relation formulae at time Tn; i.e., given timed state $[I, S, Tn]$, where $Tn(I)$ and $Tn(S)$ are corresponding DLTS time points represented by equivalence class time region Tn, tlc/6 computes whether or not $\langle\langle I, Tn(I)\rangle, \langle S, Tn(S)\rangle\rangle \in \widehat{\mathcal{LC}^t}$. If so, the query succeeds and TLC holds; if not, the query fails and TLC does not hold. The input parameter Matched is a list of [Sigma,S_] tuples representing specification output actions Sigma and specification to-states S_ that are matched earlier from implementation location I at some time Tm preceding time Tn. The to-states S_ are recorded and checked to insure that behaviorally distinct to-states are not overlooked. The input parameter UnMatched is a list of [Sigma,S_] tuples representing specification output actions Sigma and specification to-states S_ that have not yet been matched by the implementation from location I for any time Tm preceding time Tn. Lists Matched and UnMatched allow specification outputs to occur earlier or later than implementation outputs. If the implementation cannot match a specification output Sigma, TLCS adds the specification output Sigma and reached location S_ to UnMatched, increments time to the next time value Tn_ and checks to see if tlc(I,S,Tn_,Matched,[[Sigma,S_]|UnMatched],Parent) holds[2]. When the implementation matches [Sigma,S_], then [Sigma,S_] is removed from UnMatched, and added to Matched. When time passes all clock bounds (i.e., Tn=Tn_) or the implementation location invariant expires, UnMatched must be empty, or else tlc/6 fails (i.e., the implementation never matched some specification output). The input parameter Parent is a timed state number (integer) that TLCS assigns

[1] The name of a Prolog query is the Prolog atom preceding the left parenthesis. A Prolog functor is the query name and number of query arguments separated by a forward slash.

[2] The symbol | stands for list construction; e.g., [a|L] prepends the atom a on list L and [[a,b,c]|L] prepends the triple [[a,b,c]|L] on list L.

to uniquely identify the time state `['I,'S,'Tn]` that led to `[I,S,Tn]` by some action or time progression. TLCS uses timed state numbers to generate error traces and simulations.

Figure 12 displays the queries from the tlc/6 procedure that implement the TLC definition (Def. 29) formulae. The query names are derived from the system whose actions are being matched and the type of action that is being matched. Alphas are inputs, betas are outputs, and taus are τ's. For example, `spec_alpha/9` implements checking that specification inputs are matched according to Formula 22. The parameters `IActs` and `SActs` are lists of `[Sigma,Resets,ToState]` triples denoting locations reached from the implementation and specification states `I` and `S` respectively at time `T` by action `Sigma`$\in$ *Act*; `Resets` denote the set of `I` or `S` clocks reset when the transition is taken to `ToState`. `N` is the unique number TLCS assigns to timed state `[I,S,T]`.

```
spec_alpha(I,S,T,SActs,IActs,Matched,UnMatched,Parent,N),
spec_beta(I,S,T,SActs,IActs,Matched,UnMatched,Parent,N),
spec_tau(I,S,T,SActs,IActs,Matched,UnMatched,Parent,N),
imp_beta(I,S,T,IActs,SActs,Matched,UnMatched,Parent,N),
imp_alpha(I,S,T,IActs,SActs,Parent,N),
imp_tau(I,S,T,IActs,SActs,Matched,UnMatched,Parent,N),
```

Figure 12. TLC Formulae Queries.

5.3.2 Checking TLC Formulae. The spec_beta/9 procedure requires allowing both structural and temporal differences and provides the most thorough example. Figure 13 is the TLCS Spec-Beta Procedure implementing TLC Def. 29 Formula 23. Identifiers that are capitalized are Prolog variables, `[]` denotes the empty list, and underscores without prepended atoms are "don't care" variables.

The five different Prolog **rules** separated by periods in Figure 13 define the spec_beta/9 procedure. The first four spec_beta/9 rules can satisfy Formula 23. At runtime, Prolog checks the rules in the order they are defined from the top to the bottom. The fifth spec_beta/9 rule never succeeds (`fail` cannot succeed); it asserts a deficient fact (d/6) to assist in debugging TLC failures and then fails. The exclamation points are "cuts" that stop Prolog from trying to satisfy

```
spec_beta(_,_,_,[],_,_,_,_,_) :- !.
spec_beta(I,S,T,[[Beta,_,_]|SActs],IActs,Matched,UnMatched,P,N) :-
   not output(Beta),
   !,
   spec_beta(I,S,T,SActs,IActs,Matched,UnMatched,P,N).
spec_beta(I,S,T,[[Beta,SResets,S_]|SActs],IActs,Matched,UnMatched,P,N) :-
   member([Beta,IResets,I_],IActs),
   reset_time(IResets,SResets,T,T_),
   tlc(I_,S_,T_,[],[],N),
   !,
   spec_beta(I,S,T,SActs,IActs,Matched,UnMatched,P,N).
spec_beta(I,S,T,[[Beta,_,S_]|SActs],IActs,Matched,UnMatched,P,N) :-
   % Must do output before another visible action occurs--via tau or delta.
   merge_set([[Beta,S_]],UnMatched,UM),
   spec_beta_aux(I,S,T,IActs,Matched,UM,N),
   !,
   spec_beta(I,S,T,SActs,IActs,Matched,UnMatched,P,N).
spec_beta(I,S,T,[[Beta,_,S_]|_],_,_,_,P,N) :-
   retractall(c(I,S,T)),
   assert(d(I,S,T,sb(Beta),P,N)),
   fail.
```

Figure 13. TLCS Spec-Beta Procedure.

spec_beta/9 with more than one rule. Once queries in a rule body are satisfied to the "!", the remaining queries in the rule body must succeed or spec_beta/9 fails regardless of the remaining rules.

The first rule satisfies the query when there are no `[Beta,S_]` action and to-state pairs possible from `S`. This is the case when `S` cannot do any actions, or when all of `S`'s outputs have been checked.

The second rule satisfies the query when the action Beta is not an output. This is the case when the list of actions `SActs` has a `[Beta,SResets,S_]` triple and `Beta` is not an output action.

The third rule satisfies the query when the implementation matches the specification's `Beta` and the to-locations of the two systems also satisfy tlc/6.

The fourth rule satisfies the spec_beta query if it finds a matching output by following an implementation tau or by allowing time to pass. In either case, it adds the unmatched `[Beta,S_]` pair to the UnMatched list and calls spec_beta_aux/6 which is defined in Figure 14.

```
spec_beta_aux(_,S,T,IActs,Matched,UM,N) :-
   % Via tau?
   member([A,IResets,I_],IActs),
   tau(A),
   reset_time(IResets,[],T,T_),
   tlc(I_,S,T_,Matched,UM,N),
   !.
spec_beta_aux(I,S,T,_,Matched,UM,N) :-
   % Via delta?
   next_time(T,T_),
   tlc(I,S,T_,Matched,UM,N).
```

Figure 14. TLCS Spec-Beta-Aux Procedure.

Functor spec_beta_aux/6 is defined by two rules. The first rule is satisfied when the implementation can do a tau action that leads to a TLC-satisfying state. It compares the current specification state S against the tau derivative state I_ by resetting the clocks associated with I's tau action and calling tlc(I_,S,T_,Matched,UM,N). The second rule advances the time region to the next possible time equivalence class and calls tlc(I,S,T_,Matched,UM,N) to see if I matches S's output in the future. This completes the explanation of TLCS's spec_beta/9 formula.

The TLCS implementation of the remaining TLC formulae are all simpler than spec_beta/9. The only novelty is the implementation of the extra conjunct in Formula 24's antecedent which is implemented by the Prolog rule:

```
imp_alpha(I,S,T,[[Alpha,_,_]|IActs],SActs,P,N) :-
   not member([Alpha,_,_],SActs),
   !,
   imp_alpha(I,S,T,IActs,SActs,P,N).
```

The above rule simply satisfies the imp_alpha/7 query when Alpha is not in the set of actions possible by the specification.

5.3.3 Temporally Checking TLC. After checking that all TLC formulae hold for all of the actions and to-locations in IActs and SActs, tlc/6 does four things:

1. Creates the list AllMatched by adding specification outputs matched in the current timed state to Matched.

2. Creates the list StillUnMatched by removing specification outputs matched in the current timed state from UnMatched.

3. Calls next_time(T,T_) to increment the time region T to the next possible time equivalence class T_.

4. Checks to see if [I,S,T_] satisfies TLC.

When T=T_ all clocks have exceeded their maximum time bound and time progresses infinitely. Since [I,S,T]=[I,S,T_], all future behavior from locations I and S are already verified. If there are no unmatched specification outputs, TLCS asserts h(I,S,T,AllMatched) to log the fact that TLC holds.

When T≠T_, T_ is a new time region, and TLC must be verified to hold in the future states. There are four possibilities.

1. Neither I nor S can move forward to time T_. This is the case when both location invariants are violated by time T_. In this case, as long as there are no unmatched specification outputs TLC holds in timed state [I,S,T] and TLCS asserts h(I,S,T,EM) to log this fact.

2. T_ is valid for I but not for S. This means that S's invariant is violated by time T_. In this case, TLCS uses the code fragment:

```
(no_future_imp_outputs(I,T_)
 ;
 (member([Tau,SResets,S_],SActs),
  tau(Tau),
  reset_time([],SResets,T,T2),
  tlc(I,S_,T2,AllMatched,StillUnMatched,N))
 ;
 (retractall(c(I,S,T)),assert(d(I,S,T,[delta,s],Parent,N)),fail))
```

   to execute one of three things (the ";" operator is Prolog disjunction):

   (a) TLC succeeds if the implementation has no future outputs (no_future_imp_outputs/2 succeeds).

(b) TLC succeeds if I does have future outputs and they are matched by S after it performs a tau action; i.e., tlc(I,S_,T2,AllMatched,StillUnMatched,N) succeeds.

(c) TLCS retracts c(I,S,T) (the considering fact), asserts a debugging fact (d/6), and fails.

3. T_ is not valid for I but is valid for S. This means that I's invariant is violated by time T_. In this case, TLCS uses the code fragment:

```
((StillUnMatched == [],
  no_new_future_spec_actions(S,SActs,AllMatched,T_),
  (IActs == []
   ;(member([Beta,_,_],IActs),
     output(Beta)))
 ;
 (member([Tau,IResets,I_],IActs),
  tau(Tau),
  reset_time(IResets,[],T,T2),
  tlc(I_,S,T2,AllMatched,StillUnMatched,N))
 ;
 (retractall(c(I,S,T)),assert(d(I,S,T,[delta,i],Parent,N)),fail))
```

to execute one of three things:

(a) TLC is satisfied when output-bound (Def. 28) holds. Output bound is implemented here by checking three things:

i. StillUnMatched == [] verifies there are no unmatched specification outputs.

ii. no_new_future_spec_actions(S,SActs,AllMatched,T_) verifies S has no new outputs or inputs in the future.

iii. Either a previous implementation state already matched all specification actions (IActs == []) or this location is the one that matches the specification and does the output (member([Beta,_,_],IActs) and output(Beta)).

(b) If the specification does have future actions they are matched by the implementation after it performs a tau action; i.e., tlc(I_,S,T2,AllMatched,StillUnMatched,N) succeeds.

(c) TLCS retracts the considering fact c(I,S,T), asserts a debugging fact (d/6), and fails.

4. T_ is valid for both I and S. Whether or not TLC is satisfied in the future is determined by the query `tlc(I,S,T_,NewMatches,StillUnMatched,N)`. If not, TLCS retracts the considering fact `c(I,S,T)`, asserts a debugging fact `d(I,S,T,delta,Parent,N)` and fails.

When all of the tlc/6 queries from new state pairs or new time derivatives have been satisfied, tlc/6 succeeds. Otherwise tlc/6 fails. TLCS users do not input tlc/6 queries, rather, they interface with the TLC decision procedure using a tlc/2 query that accepts two TSA names or definitions and reports whether or not TLC holds between them. The tlc/2 query and other TLCS user interface queries are explained in the next section.

5.4 TLCS User Interface

This section describes the user-level data structures, Prolog queries, and outputs from the TLCS system. After describing how to define TSA for TLCS, the tlc/2 query and debugging interfaces are described.

5.4.1 TLCS TSA. Extended Backus-Naur Form (EBNF) productions in Def. 35 define the syntax of Timed Logic Conformance System (TLCS) TSA queries. The symbols "⟦" and "⟧" in the productions group optional constructs. Parenthesis "(", ")", "[" and "]" in these productions are literal. Non-terminals start with uppercase letters, and terminals start with lowercase letters. The terminal *identifier* is an alphanumeric Prolog atom.

Definition 35. TLCS TSA Query Syntax.

Tsa ::= *tsa(TSAName,[Locations,StartName,Relation])*
    Action ::= *Tau* | *VisibleAction*
    CCL ::= [] | [*ClockConstraint* ⟦, *ClockConstraint*⟧∗]
    ClockConstraint ::= [*ClockName,RelationalOperator,integer*]
    ClockName ::= *identifier*
    FromLocation ::= *StateName*
    Input ::= *identifier*
    Locations ::= [*State-CCL-Pair* ⟦, *State-CCL-Pair*⟧∗]
    Output ::= *identifier_*
    RelationalOperator ::= *l* | *leq* | *geq* | *g*
    Relation ::= [] | [*Transition* ⟦, *Transition*⟧∗]
    Resets ::= [] | [*ClockName* ⟦, *Clockname*⟧∗]

StartName ::= StateName
State-CCl-Pair ::= [StateName,CCL]
StateName ::= identifier
TSAName ::= StateName | [StateName ⟦, TimeParameter⟧*]
Tau ::= t | t(VisibleAction)
TimeParameter ::= integer
ToLocation ::= StateName
Transition ::= [FromLocation,Action,CCL,Resets,ToLocation]
VisibleAction ::= Input | Output

A TLCS TSA is a 3-tuple `[Locations,StartName,Relation]` where:

- `Locations`: a list of `[LocationName,CCL]` pairs where `CCL` is a sorted past-closed Clock Constraint List; e.g., $a < 6$ & $b \leq 4$ is encoded by `CCL = [[a,l,6],[b,leq,4]]`.
- `StartName`: Name of the starting location, either an atom or string corresponding to one of the `LocationName`'s in `Locations`.
- `Relation`: the transition relation 5-tuple list: `[[F,Sigma,CCL,Resets,T],...]` where:
  - `F,T` : location names (atoms or strings from `Locations` tuples).
  - `Sigma` : action (e.g., `a,b` = inputs, `a_,b_` = outputs, `t` = tau).
  - `CCL` : sorted Clock Constraint List.
  - `Resets` : a sorted list (set) of clock names to reset.

TSA clock constraints are specified using non-negative integers, but TSA clocks are real-valued and TSA model system behavior over the positive real valued n-dimensional continuum (for an n-clock TSA).

The tsa/2 predicate shown in Figure 15 defines a 4-location inverter TSA with minimum (`MinD`) and maximum (`MaxD`) time delays on its response to input events. The inverter clockname is `k`. Inverter locations are `inv00`, `inv01`, `inv10`, and `inv11` specifying the value of the input and outputs in each of the 4 locations. Possible initial locations are `inv01` and `inv10` (inverter's stable locations). A stable location is a location from which no output or internal action transition is defined. The inverter input label is `a`, and its output is labeled `b_`.

```
tsa([Inv,MinD,MaxD],
    [[[inv00,[[k,leq,MaxD]]],[inv01,[]],[inv10,[]],[inv11,[[k,leq,MaxD]]]],
     Inv,
     [[inv00,a, [[k,l,MinD]],[],inv10],
      [inv00,b_,[[k,geq,MinD]],[], inv01],
      [inv01,a, [],[k],inv11],
      [inv10,a, [],[k],inv00],
      [inv11,a, [[k,l,MinD]],[],inv01],
      [inv11,b_,[[k,geq,MinD]],[], inv10]]]) :-
      member(Inv,[inv01,inv10]).
```

Figure 15. TLCS Inertial *Inverter.*

Given the inverter definition in Figure 15, the Prolog query `tsa([inv01,2,3],X).` returns the three-tuple TSA:

```
X = [[[inv00,[[c,leq,3]]], [inv01,[]], [inv10,[]], [inv11,[[c,leq,3]]] ],
     inv01,
     [[inv00,a,[[c,l,2]],[],inv10], [inv00,b_,[[c,geq,2]],[],inv01],
      [inv01,a,[],[c],inv11], [inv10,a,[],[c],inv00],
      [inv11,a,[[c,l,2]],[],inv01], [inv11,b_,[[c,geq,2]],[],inv10] ]]
```

5.4.2 TLCS Parallel Composition. TLCS also parallel composes TSA to generate models of more complex systems. EBNF productions in Def. 36 define the syntax of TLCS parallel composition queries. These productions rely on those productions already specified in Def. 35.

Definition 36. TLCS Parallel Composition Query.

Parallel-TSA-Composition ::= *parallel(TSAList,Hidden,PTSA)*
ActionPair ::= *[NewAction,OldAction]*
Hidden ::= [] | *[VisibleAction* ⟦, *VisibleAction*⟧*]
NewAction ::= *VisibleAction*
OldAction ::= *VisibleAction*
PTSA ::= *[Locations,StartName,Relation]*
Renames ::= [] | *[ActionPair* ⟦, *ActionPair*⟧*]
TSAId ::= *TSAName* | *Tsa*
TSAList ::= *[[TSAId,Renames]* ⟦, *[TSAId,Renames]*⟧*]

The query `parallel(TSAList,Hidden,PTSA)` parallel composes together the TSA in `TSAList` where:

- `TSAList` is the input list of TSA names or definitions and renaming tuples For example, the list `[[tsa1,[[newsig1,oldsig1],...]]...]` renames `tsa1`'s labels `oldsig`$_i$ to `newsig`$_i$.

- Hidden is the input list of uncomplemented (no trailing underscores) actions; parallel/3 generates taus according to the actions in the list Hidden. The generated taus hide internal actions of the parallel TSA so that they are not available for interaction with TSA outside of the parallel composition (i.e., hidden actions make the parallel TSA a black box that can only be accessed using its unhidden actions).

- PTSA is the returned three-tuple TSA, the location names are vectors of location names from the component location names unless the query "state_vectors." is executed to toggle TLCS to use abbreviated location names.

For example, the query:

```
parallel([[[and000,1,2],[[ab,c]]],
          [[inv01,1,2],[[ab,a],[c,b]]]],
         [ab],
         Nand).
```

returns 3-tuple TSA Nand, which is the parallel composition of TSA and000 with minimum and maximum delay 1 and 2 and TSA inv01 also with minimum and maximum delay 1 and 2. Renaming and000's c output to ab and inv01's a input to ab connects them and names Nand's internal signal ab. Since ab is restricted, it will appear in TLCS traces as t(ab). Nand's output is c_, accomplished by renaming the inverter's b output to c.

Parallel/3 generates the reachable location space by starting from the initial location of each subcomponent and generating transitions and new to-locations according to the TSA parallel composition rules defined in Def. 18 on page 45. Parallel/3 stops generating transitions and to-locations when no new transitions are possible. Parallel/3 generates the reachable location space with regard to the cooperating actions of the TSA being composed, but it does not eliminate location combinations that might actually be unreachable because of impossible clock combinations. Even though parallel/3 generates temporally impossible location combinations, only timed states reachable under the given clock conditions are examined by tlc/6.

5.4.3 TLC Query. Given the Section 5.4.1 inverter TSA, the TLCS command-line entry "`tsa([inv01,2,3],X),tsa([inv10,2,3],Y),tlc(X,Y).`" returns yes because even though the TSA `X` and `Y` do not have the same initial location names, their behavior is identical. This means that `X` is an acceptable implementation of `Y`. Since they are identical, `Y` is also an acceptable implementation of `X`, and the query "`tsa([inv01,2,3],X),tsa([inv10,2,3],Y),tlc(Y,X).`" also returns yes. However, the query "`tsa([inv01,2,4],X),tsa([inv01,2,3],Y),tlc(X,Y).`" fails returning the diagnostic information:

```
The first deficiency discovered was:
I:inv00
S:inv00
T:[[[k0, 3, 3], [k1, 3, 3]], [[k0, k1]]]
Prob:[delta, s] P#:26 M#:27
Where possible Imp Actions are: [b_]
and possible Spec Actions are:  [b_]
and future Imp outputs are not matched by the Spec.
```

In this case, both the implementation and specification are in location `inv00`; the time region `T=[[[k0, 3, 3], [k1, 3, 3]], [[k0, k1]]]` where both clocks `k0` and `k1` are at time 3 (i.e., `CV=[[k0, 3, 3], [k1, 3, 3]]`), and their fractional parts are equal (`k0`, and `k1` are in the same partition element `CC=[[k0, k1]]`); the problem (`Prob`) is with time progressing (`delta`) in the specification (`s`); the parent timed state is #26 and this timed state is #27. The problem is that time cannot progress any more in the specification location `inv00`, but implementation location `inv00` can continue producing future `b_` outputs. Hence, implementation `b_` outputs are not a timed subset of the specification `b_` outputs, and TLC fails to hold.

The subsequent query "`trace_to.`" returns the trace

```
=a=4=b_=a=6==>
```

After inputting an `a`, passing through 4 time regions, outputting a `b_`, inputting another `a`, and passing through 6 more time regions TLCS arrives at the divergent timed states. The query "`compare_states(inv00,inv00).`" returns:

```
IInv:[[k0, leq, 4]]
```

```
A:a G:[[k0, l, 2]] R:[] I_:inv10
A:b_ G:[[k0, geq, 2]] R:[] I_:inv01

SInv:[[k1, leq, 3]]
A:a G:[[k1, l, 2]] R:[] S_:inv10
A:b_ G:[[k1, geq, 2]] R:[] S_:inv01
```

This information includes the implementation location invariant and possible implementation transitions and the specification location invariant and possible specification transitions. In this case the implementation invariant `IInv:[[k0,leq,4]]` is looser than the specification invariant `SInv:[[k1,leq,3]]`, leading to the non-TLC-satisfying behavior.

TLCS distribution includes files that define the basic monotonic and inertial gate-level primitives. There are also files defining procedures that simulate TSA (`simulate(TSA)`), pretty-print TSA (`pp_tsa(TSA)`), and print CCS agents from TSA (`ccs_agent(StateNamePrefix,TSA)`). TLCS is available via email to `f.c.young@ieee.org`. TLCS runs on the public domain SWI-Prolog available via anonymous FTP from Jan Wilemaker at `ftp.swi.psy.uva.nl/pub/SWI-Prolog`.

5.5 Summary

This chapter describes the Prolog Timed Logic Conformance System (TLCS). After presenting some background information, it describes the finite automata induced from Timed Safety Automata (TSA) called region automata. After describing the TLCS region automata time representation, the chapter describes the TLCS Prolog rules and procedures that efficiently implement the TLC decision procedure. Finally, it concludes with a description of the TLCS TSA input format, TLCS TSA parallel composition, and the TLCS user interface. The final section includes TSA syntax for TSA definitions and parallel composition, and it explains example debugging information available when TLC properties are not satisfied between two systems.

VI. Application

This chapter is devoted to showing the utility of the Timed Logic Conformance (TLC) relationship for systems engineering and verification. It describes system models and explains the results of TLC verifications at several different levels of abstraction.

The flexible time and behavior modeling capabilities of Timed Safety Automata (TSA) make it possible to express the relationship between time passing and behavior at many different levels of abstraction. Virtually any other discrete state-based modeling formalism can be mapped into TSA, including all untimed finite state machine models, timed event graphs (HB94), and timed CCS. Such a flexible modeling formalism makes it easy to model different kinds of behavior, but designers should constrain themselves to specific canonical forms for modeling behavior so that the semantics of the models can be uniformly understood and results can be applied in meaningful ways. The next few sections discuss using TSA and Time Logic Conformance System (TLCS) to canonically model and verify hardware systems at three different levels of abstraction.

6.1 Gate-level Models

Logic gates are the primitives in a gate-level hardware model. Typically a logic gate discretely models the behavior of several interconnected transistors by abstracting real-valued voltage levels into the binary values zero and one. In practice designers use tools like SPICE to analyze component models below the gate-level (e.g., individual transistors) because of their bi-directional current flows and continuous electro-magnetic properties.

6.1.1 Canonical Gate-Level Models. This section focuses on two canonical forms for modeling gates that conform to the Definition 30 modeling constraints. The first form is called **monotonic**; a monotonic gate model reflects every possible output change that can occur from all **unstable locations.** An **unstable location** is a location where a TSA may generate an output or internal event. Consequently, a **stable location** is a location where no output or internal events are possible. In contrast to the monotonic gate model, an **inertial** gate model might not reflect an output change from an unstable location. An inertial gate model output event is "canceled" when the time separation between two input events is small enough (i.e., two events occur on the same input with less than a minimum delay between them) and the model returns to a stable location before generating the output event. Monotonic semantics are the standard for untimed gate models. In timed systems, inertial models support higher fidelity modeling, as shown in Section 6.1.2.

Inertial-delay semantics are commonly used in hardware simulations, and they are the default signal assignment semantics of the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). Although inertial delay models can make simulations more efficient, they can also hide defects in systems if not used correctly. In TLCS, as shown in the following examples, inertial gate models model unstable-location dependencies that are important to investigate for proper implementation behavior. Since the inertial delay gates model hardware characteristics in more detail than monotonic gates, inertial gates are used in most of the following hardware examples. Inertial delays are modeled with minimum and maximum bounds, not just a single delay value, further enriching the model's fidelity in accordance with accepted practice (BS91, Bur92).

The next section examines some simple TSA models of hardware primitives that can be used to build larger systems by parallel composition. It also examines a simple abstract specification and the results of comparing implementations against specifications.

6.1.2 Inverters, Ands, and Nands. The simplest hardware device modeled in this research is an *Inverter.* Figure 16 displays the logic symbol and the TSA defining the behavior of a monotonic *Inverter.* The *Inverter* clock name is k. In the figure, black triangles (▶) touch stable TSA locations, and unstable locations have no triangles. Note that the *Inverter* can be configured to start in either stable location. *Inverter* locations are labeled with the two-digit binary codes indicating the values of the *Inverter* input and output in that location. The *Inverter* is monotonic because after entering an unstable location (i.e., locations 00 or 11), inputs that would return the device to a stable location (called **stabilizing inputs**) are not allowed, and an output event will occur. Only those actions explicitly specified as TSA transitions are possible. In parallel compositions attempted inputs to the inverter while in locations 11 and 00 violate the CI-free property.

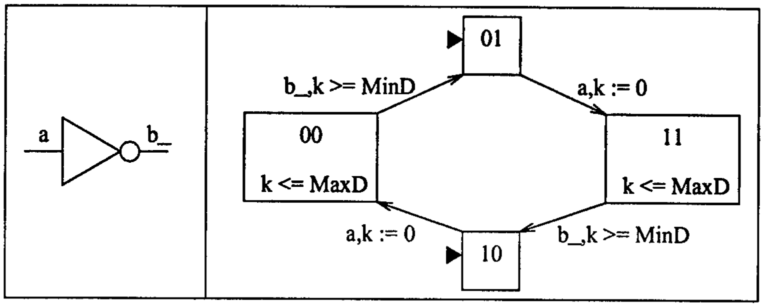


Figure 16. Monotonic *Inverter* Logic Symbol and TSA.

Figure 17 is another TSA defining the behavior of a inertial *Inverter*. Figure 17 is identical to the TSA in Figure 16, except that it includes two additional transitions from the unstable locations that model input changes occurring before the output actions. Hence, a spike on the a input to the inertial *Inverter* may occur and not generate a b₋ output action; this *Inverter* has inertial-delay semantics during the interval [0,MinD). In practice, it might be the case that an even smaller inertial time period, and not the whole time interval [0,MinD) would be better for high fidelity modeling because it models the inertial and unreliable states of a circuit explicitly. Such a model can be accommodated by adding another timing parameter to the TSA. For simplicity, and in agreement with the general bi-bounded delay model (BS91, Bur92) used in all of the related work, more detailed models are not described here.

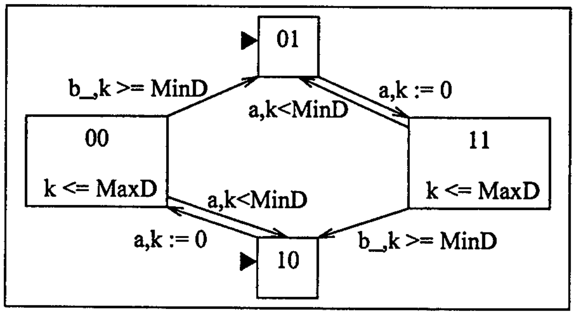


Figure 17. Inertial *Inverter* TSA.

The next simple hardware primitive is a two-input *And.* Figure 18 depicts the logic symbol and the TSA defining the behavior of an inertial two-input *And.* In unstable locations, until the minimum delay has passed, stabilizing inputs can occur, so the gate has inertial-delay semantics during the interval [0,MinD). In this model, locations are labeled with three-digit binary codes indicating the values of the *And*'s two inputs and output in that location. For example, the location 101 is an unstable location, where input a is asserted, and input b is de-asserted, and output c_ is asserted. Note that all eight possible combinations of three boolean variables are represented, so the model is at a detailed level of abstraction. Also note that every TSA input action from a stable to unstable location resets k and that every unstable location has the invariant $k \leq$ MaxD for some integral delay MaxD. *And* can start from any stable location.

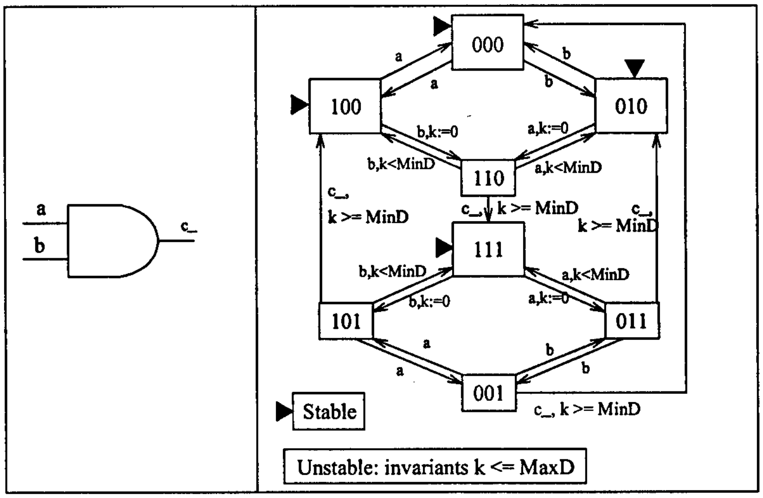


Figure 18. Two-Input *And* Logic Symbol and TSA.

Figure 19 depicts the logic symbol and Timed Safety Automata (TSA) defining the behavior of an inertial two-input *Nand.* A *Nand* is very similar to an *And,* except for the fact that the *And* stable/unstable locations are swapped to *Nand* unstable/stable locations. Hence, the location invariants are swapped from the unstable and-locations to the unstable nand-locations, and transitions are reversed.

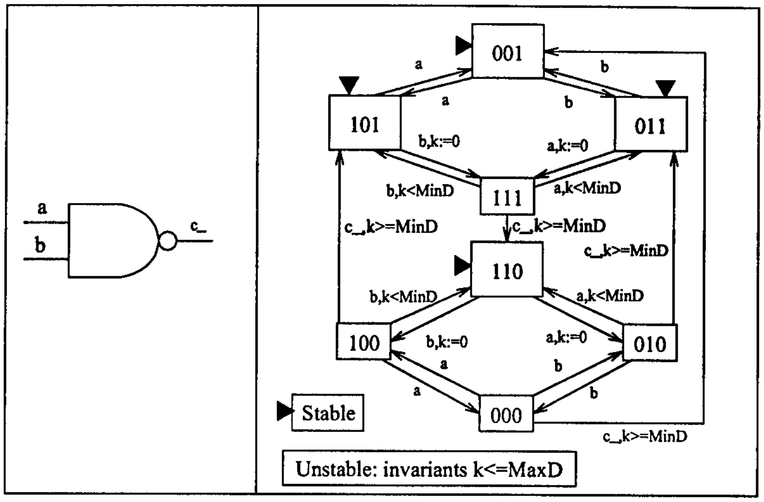


Figure 19. Two-Input *Nand* Logic Symbol and TSA.

One of the simplest *And* implementations is to couple a *Nand* and *Inverter* together as shown in Figure 20. Figure 21 is the TLCS definition of a *Nand* and *Inverter* composed in parallel. The nand001 `c_` output is renamed to `mid_`, and the *Inverter* `a` input is renamed to `mid`, and the *Inverter* output `b_` is renamed to `c_` to match the action labels of the 2-input *And* in Figure 18, and the resulting `mid_` action is hidden, changing it to a τ(`mid_`).

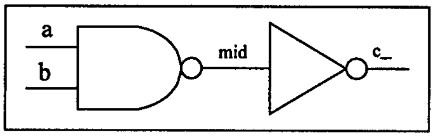


Figure 20. *Nand/Inverter And* Implementation Circuit Diagram.

```
parallel([[[nand001,NandMin,NandMax],[[mid,c]]],
          [[inv10,InvMin,InvMax],[[mid,a],[c,b]]]],
         [mid],
         PAnd)
```

Figure 21. Parallel *Nand* and *Inverter And* Implementation.

Depending on the timing of the gates, this parallel *And* is an acceptable implementation of the *And* "specification" in Figure 18. Comparing the timing relationships TLC accepts is interesting. Generally, given *And*'s minimum and maximum delays AndMin and AndMax, one expects that the timing relationship is satisfied whenever NandMin + InvMin $\geq$ AndMin and NandMax + InvMax

≤ AndMax. That is the case when monotonic gates are used, but that is not the case with inertial gates! With inertial gate models, the parallel-and implementation can output a c₋ earlier than the *And* specification allows when NandMin + InvMin = AndMin.

For example, assume that the *Nand* and *Inverter* minimum and maximum delays are 1 and 2 time units, and that the *And* specification minimum and maximum delays are 2 and 4 time units respectively. Imagine the implementation and specification inputs wired in parallel together as diagrammed in Figure 22, and refer to the timing diagram in Figure 23 during the following discussion.

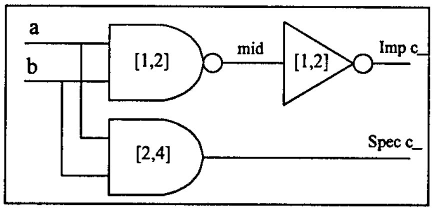


Figure 22. *And* Implementation and Specification in Parallel.

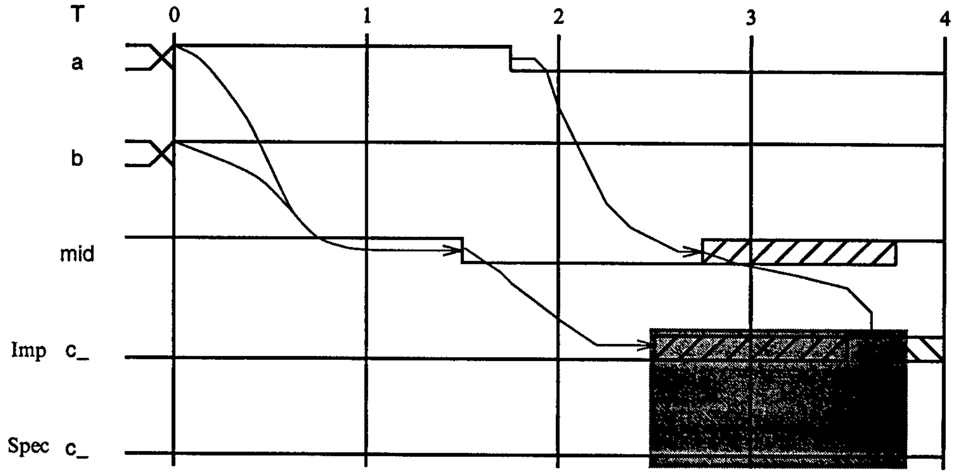


Figure 23. *And* Implementation-Specification Timing Diagram.

Let T be the point of reference for time passing, and let T = 0 just when the last input is asserted from 0 to 1. Then at T = 1.5 the TSA can be in locations implementation:[nand111, inv10], and specification:and110, where both the implementation and specification are in unstable locations. Then, τ(mid_) de-asserts (changes from 1 to 0) and moves the TSA to locations implementation:[nand110, inv00], and specification:and110 where only the *Nand* TSA is in a stable location. If another a input occurs at $T = 1.75$, before the *And* can assert its output, then the *And* specification stabilizes in state and010, and the *Nand* destabilizes to nand010. Eventually, if no more inputs occur before $T = 3.75$, the *Nand* will assert τ(mid_)and stabilize in state nand011. Until the *Nand* stabilizes, the *Inverter* is still unstable in state inv00, and it can generate a c_ output for $T \in [2.5, 3.5]$. The specification cannot generate this c_ output. This difference between the specification and implementation outputs is highlighted by the shaded area in Figure 23. If, however, timing parameters are changed such that the specification's minimum delay (AndMin) is 1 time unit or less, and its maximum delay is 4 or more, and the nand and *Inverter* delays are bounded by 1 and 2 time units, TLC is satisfied because the implementation cannot then produce any outputs outside the time bounds allowed by the specification.

In general, for inertial gates with non-zero gate delays, given that PMin $\triangleq$ NandMin + InvMin and PMax $\triangleq$ NandMax + InvMax, TLC holds whenever PMin $\geq$ AndMin $\wedge$ PMax $\leq$ AndMax $\wedge$ AndMin $\leq$ NandMin. Note for example that a *Nand* with delays in $[2, 3]$ and *Inverter* with delays in $[1, 2]$ satisfies an *And* specification with delays in $[2, 6]$, but switching the delays—i.e., *Nand*: $[1, 2]$ and *Inverter*: $[2, 3]$, fails TLC with the *And* delay $[2, 4]$ because the unstable *Inverter* can still assert c_ earlier than allowed by the specification as shown above.

Verification results are very dependent on the models chosen as illustrated in this example. In particular, TLC verification results are different for the monotonic and inertial gate models. Some of the most difficult errors to track down in hardware devices are those associated with unstable states; since it is important to create designs that do not suffer from obscure defects like this, inertial gate models are used for this research.

6.1.3 Gate-Level Model Summary. The previous section described two different types of gate-level models: monotonic and inertial. In unstable states, monotonic gates do not allow any stabilizing inputs and always reflect the pending output or internal action. In contrast, inertial gates allow stabilizing inputs in the interval [0,MinD) after the gate entered the unstable state. These stabilizing inputs cancel the pending internal and output actions from those unstable states and more accurately model the behavior of real gates. The more detailed inertial models are better suited for discovering and correcting obscure timing defects. Although even more detailed models are possible, the inertial gates are accurate enough to find real problems and simple enough to support efficient computation of the TLC relation.

6.2 Asynchronous Hardware Components

Many hardware designers may believe that "faster implementations are always better," but, for many low-power and asynchronous designs, specification minimum delays must be verified as well. In the case of the *Nand- Inverter* and *And* implementation example from Section 6.1.2, the device to which the *And* output is connected will likely be sensitive to the *Nand-Inverter*'s unexpected output, so designers must ensure that the implementation's outputs are a timed subset of the specification as checked by TLCS. Unstable-location outputs like those from location [nand010,inv00] are very difficult to anticipate and test for in actual circuits; hence, the utility of the TLC relation and decision process to root out inconsistencies between specifications and implementations. The ability to richly model timing dependencies like this is especially important for hardware engineers working with high-performance synchronous and asynchronous designs. However, since protocol-dependent asynchronous design allows more variations and poses a more difficult verification challenge, the next section focuses on commonly used asynchronous components composed of several gates.

A short discussion about hazards precedes the first asynchronous hardware component example. A **hazard** is a problem associated with hardware circuits.

6.2.1 Hazards. Asynchronous hardware engineers have used different hazard models to analyze problems with sequential and combinational circuits. Stevens formalized the following assumptions, hazard models, and hazards (Ste94):

Definition 37. Hazard Assumptions.

- **Fundamental Mode Assumption** *The environment is constrained to hold the inputs stable long enough to allow the changes to propagate through the logic, produce the desired results, and stabilize internally before changing the inputs again.*

- **Isochronous Fork Assumption** *The difference between delays on the different sections (called* **forks***) of connected wires is insignificant, hence the change in value of a wire is propagated to, and reaches, all devices connected to that wire simultaneously.*

Unlike untimed FSM-based logics, TSA models do not constrain designers to adopt either of the above assumptions. Assumptions are specified in the TSA models by their construction. For example, a non-fundamental mode TSA can be constructed by explicitly modeling all transitions from every location, for all possible times, for every possible input. Sometimes designers might desire to do this and include error states for disallowed input sequences or combinations. Typically however, this is too tedious, so designers may abstract the behavior by leaving out disallowed inputs, and relying on TLC to tell them when fundamental mode assumptions are violated.

The timed nature of the TSA implicitly specifies *when* fundamental mode assumptions are being made by explicitly declaring the times when inputs are possible. TLCS ensures that the fundamental mode assumptions made by a specification are not violated by TLC satisfying implementations, because as soon as a specification TSA is ready for input, the implementation TSA must also be ready to accept the same input. TSA are also powerful enough to model isochronous and non-isochronous systems; a non-isochronous situation can be modeled by modeling each wire segment of the system as a buffer-like TSA with its own clock.

Generally, the TSA used for this research are isochronous and multiple-input-change (MIC)-capable, and they explicitly specify the fundamental mode assumptions made. Unfortunately, the rapid linear increase in the number of clocks, whether the clocks come from the number of devices or from modeling wires as TSA, exponentially explodes the number of possible timing relationships and can make it impractical to reason about even small circuits with today's computers. Consequently, although TSA provide the capability to model systems in as much detail as required, one must always choose between model fidelity and practicality.

Given Definition 37, four hazard models can be defined. A hazard model is a set of conditions that describes the level of detail a designer uses to analyze a design.

Definition 38. Hazard Models.

- **Delay Insensitive (DI) Model** *Both device and wire delays are considered, no isochronous fork assumptions are made. This is the most detailed and accurate model because no timing assumptions apply.*

- **Quasi Delay Insensitive (QDI) Model** *Some of the forked interconnections must be isochronous for circuits to be hazard-free. This model makes timing assumptions about some of the wire forks in the design.*

- **Speed Independent (SI) Model** *All of the forked interconnects are isochronous, and interconnect delays are lumped into device delays. This model makes timing assumptions about all wires in the design.*

- **Burst Mode Model** *Inputs and outputs are mutually exclusive and the device must stabilize before subsequent inputs arrive. Hence, a burst mode compliant device will only generate an output* after *all inputs in a given input sequence have arrived, and it will not subsequently change its output unless more input changes occur. No new inputs are allowed until the circuit has stabilized. Burst mode is the most abstract of the four hazard models, and it can be used* *with either DI, QDI, or SI wire models, but the burst mode model is typically used with the SI wire model.*

Depending on the level of detail included in the TSA themselves, all four of the hazard models can be supported by TSA. Modeling some (in the QDI case) or all (in the DI case) wires as buffer-like TSA support the more detailed QDI and Delay Insensitive hazard models. Therefore, TLCS supports verifying that DI, QDI, SI, implementations and specifications are consistent. The SI model is used in this research along with upper and lower bounded device delays to accommodate some wire length, resistance, and capacitance variations.

Given the above hazard models, five general and three sequential-circuit hazards occur often enough to have special names and definitions.

Definition 39. General Hazards.

- **Static Zero Hazard** *A device output should be stable at zero, but it momentarily outputs one before returning to zero. This is the hazard that disqualifies the* Nand/Inverter *implementation of the* And *in the example discussed in Section 6.1.2.*

- **Static One Hazard** *A device output should be stable at one, but it momentarily outputs zero before returning to one.*

- **Dynamic Hazard** *A device output is changing to a new value, but it changes to that value more than once before stabilizing.*

- **Function Hazard** *A particular dynamic hazard that exists in a MIC circuit if and only if an output changes more than once along a minimum length path of an input transition. Function hazards cannot be removed by changing the circuit design (Ste94:56).*

- **Delay Hazard** *A hazard that is associated with devices having more than one implicant enabling a function output in any location. Because of different delays on the implicant* *paths, after the faster implicant asserts, and before the slower one asserts, subsequent inputs destabilize and de-assert the faster implicants before the slower implicant finally asserts the output.*

General hazards may occur in any circuit, either combinational or sequential, but there are three hazards explicitly defined for sequential circuits only.

Definition 40. Sequential Hazards.

- **Essential hazard** *A hazard where the device stabilizes in a different state after a single event than after three consecutive events on the same input.*
- **Transient Hazard** *A hazard that occurs from a stable state when two events on a single input return the device to the stable state, and there is a static hazard on any output. This is caused by the output logic, and not the state-holding logic of the circuit.*
- **D-Trio Hazard** *A hazard that occurs when from a stable state, three input events on a single input return the device to the state entered after the first input, and there are different outputs in any of the entered states.*

Given that a specification disallows any of the above hazard conditions, and they exist in an implementation, the TLC relation identifies those differences and TLCS reports the first one it encounters because they all represent implementation outputs that are not allowed by the specification and TLC detects those errors via Formula 23.

Several different hazards have been defined and related to the *Nand* example already presented. The next section returns to more models and the application of TLC to more complex designs.

6.2.2 C-Elements. One commonly used asynchronous circuit component is a C-element. The C-element specification is a level of abstraction above a gate-level model. Table 6 describes the output of a C-element based on its input values.

Table 6. C-element Function.

Inputs	Output
00	0
01	No Change
10	No Change
11	1

Figure 24 is the C-element logic symbol and a specification TSA. The TSA definition allows specifying minimum feedback delay (FB) in the environment from a C-element output to its next a or b input, and a minimum and maximum bound on the C-element's output response from the time of the last input. Positive FB explicitly increases the amount of time allowed for the implementation to conform to the fundamental mode assumption of the C-element. Positive FB adds to the time that passes in locations XY/X, YX/X, and YY/X before the C-element specification returns to location XX/X where it can once again accept either an a or b input.

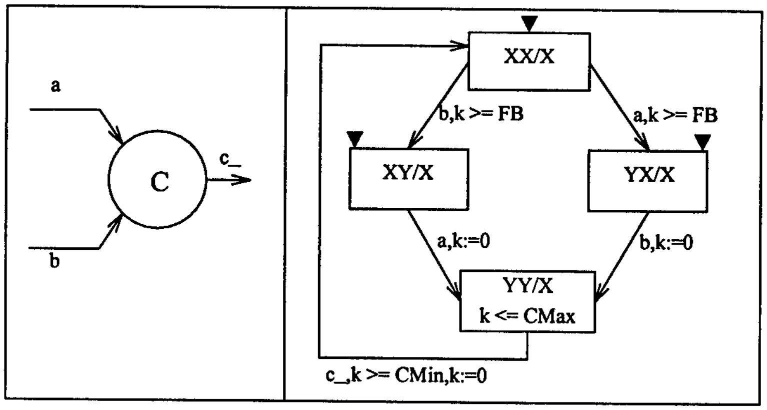


Figure 24. Two-Input C-Element Logic Symbol and TSA Specification.

The C-element specification TSA in Figure 24 is not complete because it omits the behavior of the C-element when one of its inputs changes value twice before the other input changes. Since that behavior is important in the application focused on in the next section, the missing behavior must be added to the C-Specification TSA. Figure 25 is the "wobbly" C-Specification TSA required.

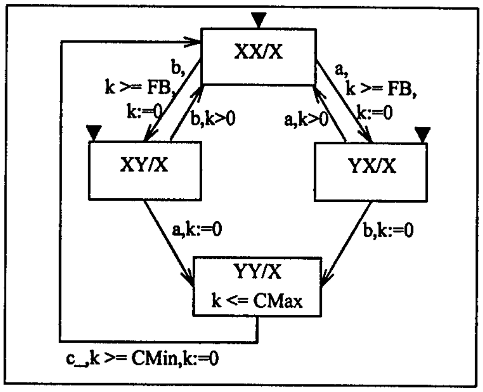


Figure 25. Wobbly Two-Input C-Specification TSA.

Earlier Figures 4 and 5 on pages 43 and 43 depict a standard (but not completely hazard-free) way to implement C-elements, and the corresponding TLCS clause defining the parallel C-element implementation.

Note that verifying this implementation of a C-element in untimed calculi, including Logic Conformance, fails except under the burst-mode hazard model (Ste94:p64). Here, using TLCS and TSA capability to implicitly and explicitly specify the fundamental and burst mode assumptions made in the specification, and the timing of both the specification and implementation components, a rigorous timing analysis can be accomplished. Which implementations will actually satisfy the more detailed timed specification without generally adopting the burst mode model can be determined. TLCS can verify when the implicit burst-mode timing assumptions hold. In particular, for non-zero gate delays, and given that PMin $\triangleq$ AndMin + OrMin and PMax $\triangleq$ AndMax + OrMax, Table 7 shows the delay value relationships and whether or not TLC holds for the different gate models. As with the *And* example, in unstable locations hazards can occur when inputs change faster than the minimum delay of the receiving component. Monotonic gates succeed only with explicit non-zero feedback allowances extending the amount of time between C-element outputs and the new input. Once again, there is a strong dependency on the relationship between the minimum specification delay and the minimum delay of the input receiving component in the implementation.

Table 7. C-element Verification Results.

FB	Delay Relationship	Gate Model	TLC
FB $= 0$	PMin $<=$ CMin $\vee$ PMax $>$ CMax	monotonic	Fails[1]
FB $= 0$	PMin $\geq$ CMin $\wedge$ PMax $\leq$ CMax	monotonic	Fails[2]
$0 <$ AndMax $\leq$ FB	PMin $>=$ CMin $\wedge$ PMax $\leq$ CMax	monotonic	Holds
FB $= 0$	PMin $<=$ CMin $\vee$ PMax $>$ CMax	inertial	Fails[1]
FB $= 0$	AndMin $\geq$ CMin $\wedge$ PMax $\leq$ CMax	inertial	Holds
Fails[1], specification time bounds too tight.			
Fails[2], Imp cannot match specification input/output.			

Note that monotonic gates succeed only with explicit non-zero feedback allowances extending the amount of time between C-element outputs and the new input. However, even with FB $= 0$, inertial gates satisfy TLC when AndMin $\geq$ CMin; once again, there is a strong dependency on the relationship between the minimum specification delay and the minimum delay of the input receiving component in the implementation.

Having described specifying and verifying C-elements, the next section discusses the next level of abstraction, a component built out of C-elements.

6.2.3 STARI Queue Stage. C-elements can be used in tandem to dual-rail encode information in the stages of asynchronous queues. FIFO queue stages are made from C-elements connected together as shown in Figure 26 (TB97). A *Nor* produces the acknowledgment signal `ack_m_`. Connecting k of these queue stages together produces a k-length queue. The k-length queue is called STARI (Self-Timed At Receiver's Input). Table 8 defines the dual-rail encoding scheme used in (TB97); the `t_n_` and `f_n_` values of the individual C-elements determine the value stored in the n^{th} queue stage; empty distinguishes between a single data item stored for a long time and two consecutive data items of the same value.

Since STARI stages are asynchronous, STARI supports connecting systems together that have some skew between the phases of the sender and receiver clocks. Generally, when queues are longer more skew can be tolerated. Given that the clock-rate of the receiver is greater than or

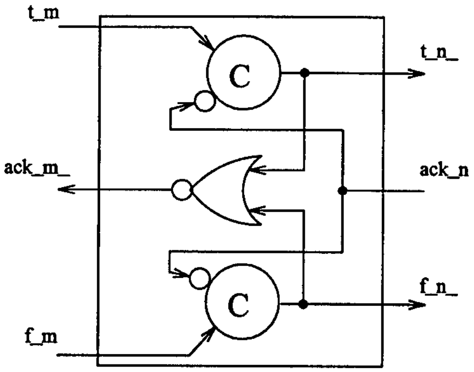


Figure 26. STARI FIFO Queue Stage.

Table 8. Dual Rail Encoding Scheme.

t_n_	f_n_	Encoded Value
0	0	empty
0	1	false
1	0	true
1	1	illegal

equal to the clock-rate of the transmitter, STARI could be used to connect systems with different clock rates, but the receiver would have to wait for the slower transmitter by watching for t_k_ and f_k_ changes. According to (TB97), correct queue operation depends on proving the following two properties:

1. "Each data value output by the transmitter must be inserted into the FIFO before the next one is output." (i.e., the transmitter must not change input values to the queue until the first FIFO stage acknowledges receiving the input by generating an ack_0_ event.)
2. "A new value must be output by the FIFO before acknowledgment from the receiver." (i.e., the receiver must not generate an ack_k_ event to the FIFO until it has received the data from the last FIFO stage.)

In (TB97), they described using the COSPAN verification system to verify STARI operation as defined in Section 2.2.3. Figure 27 is the timed process that the Berkeley researchers use as a valid abstraction of the STARI FIFO Queue Stage shown earlier in Figure 26; ▶'s indicate two potential starting locations corresponding to an empty and a full queue stage. The edges necessary to satisfy the nonblocking and stutter closure requirements are not in the diagram. This abstraction does not explicitly constrain the queue stage to legal inputs and legal output sequences; the TLCS model includes these important behavioral constraints.

The corresponding 17-location TSA is shown in Figure 28. This TSA constrains the queue stage to separate each occurrence of true and false inputs with an empty input, and it disallows the illegal dual-rail code as described in the natural language specification. Note that the inputs and outputs of the queue-stage are symbolized by the state labels. The letters T, F, and E represent true, false, and empty, and depict both the value of the inputs and output data signals; hence, the label TE depicts the situation where the stage inputs encode true, but its outputs encode empty, and the queue-stage is waiting on an `ack_n` before changing its output from empty to true. The binary labels encode the Boolean input and output values (`t_m,f_m,ack_n/t_n_,f_n_,ack_m_`). The queue stage timing parameters are [CMin, CMax, NorMin, NorMax], representing the minimum and maximum bounds on the C-element and *Nor*s used to build the queue stage.

TLCS confirms that gate-level and C-Specification-level FIFO stage models satisfy the TLC relation with STARI FIFO Queue Stage specification when consistent timing parameters are used. The gate-level FIFO stage model has 9 clocks; TLCS parallel composes six 2-input *And*s, two three-input *Or*s, and a single 2-input *Nor* to create it. The C-Specification-Level FIFO stage model has 3 clocks. TLCS composes two wobbly C-Specification models (Figure 25), and a single 2-input

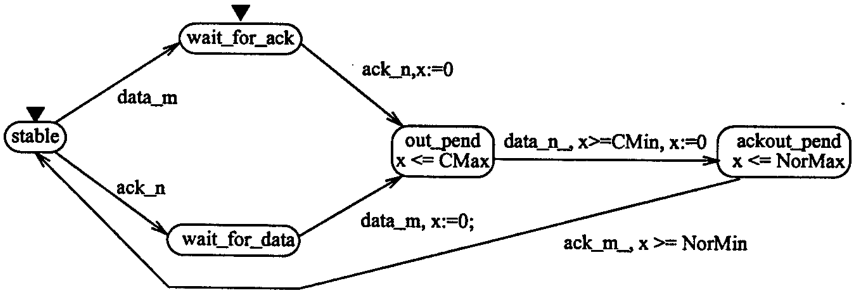


Figure 27. Abstract STARI FIFO Queue Stage Timed Process.

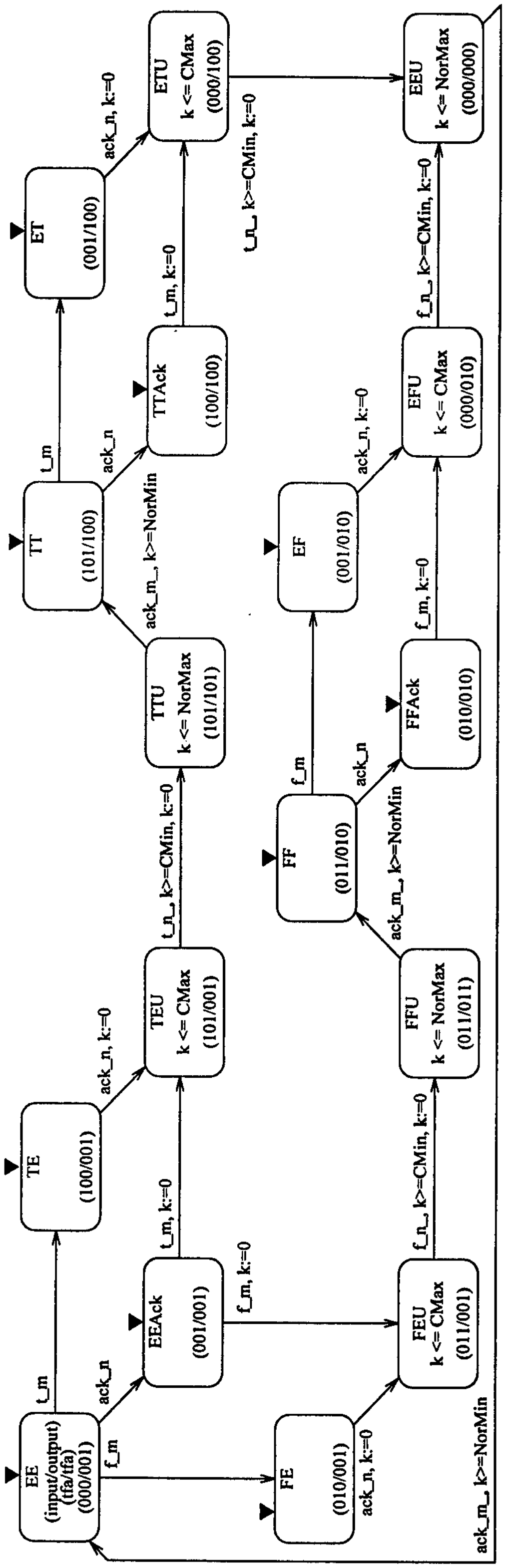


Figure 28. STARI FIFO Queue Stage TSA.

Nor generating ack_m_ as shown in Figure 26. Even though gate-delays [1,2] fail when comparing C-element implementations to C-Specifications with any delays in the previous section, gate-level FIFO stage verification succeeds with gate-delay [1,2] and C-element delays [2,4]. This is the case because the FIFO queue stage specification so constrains the C-element inputs that it does not enter the unstable states that cause the problem during the C-element/C-Specification verifications.

Recognizing where worst case instability problems are avoided (like C-elements with gate-delay [1,2] in STARI), and leveraging their absence to improve performance is a key advantage to using the TLC relation and TLCS. Of course this is done while maintaining confidence that the system is going to work correctly under all possible conditions allowed by the specification.

Notice that designers do not need to model the environment and augment TLC verification with an assumes-guarantees style reasoning process to factor in the constraints imposed on the environment for the design to work. Instead, those constraints are built into the specifications, and TLC ensures that those properties that are dependent upon the input capabilities of the specification hold in the implementation. This enables designers to rely on TLC verifications without separate models for the environment. TLC also allows implementations the freedom to accept inputs that the specification does not allow. This allows more design reuse and requires less effort during the verification and design process.

With the accurate STARI queue stage specification verified against the implementation models at two levels of abstraction, the next section discusses TLC verification at the next level of abstraction—the entire queue.

6.2.4 STARI Queue and Perfect Buffer. In the other STARI queue verifications (Gre93, BM98, TB97), researchers include models of a sender and receiver environment for the verification. They usually focus on verifying that STARI could be used to communicate between two synchronous systems operating with some clock skew between them. Figure 29 depicts the STARI queue in its environment. Note that the acknowledgment output of the queue is not connected to the transmitter. Including the environment requires assumes-guarantees style reasoning and complicates the verification process. In this configuration, researchers were obliged to prove that the environment in conjunction with the queue behave correctly by observing the actions connecting the sender to the queue and queue to receiver with two different queue models but the same sender and receiver models. TLC verification generalizes somewhat by eliminating the sender and receiver models and focusing on comparing the queue directly to a specification of its behavior—a Perfect Buffer (PB).

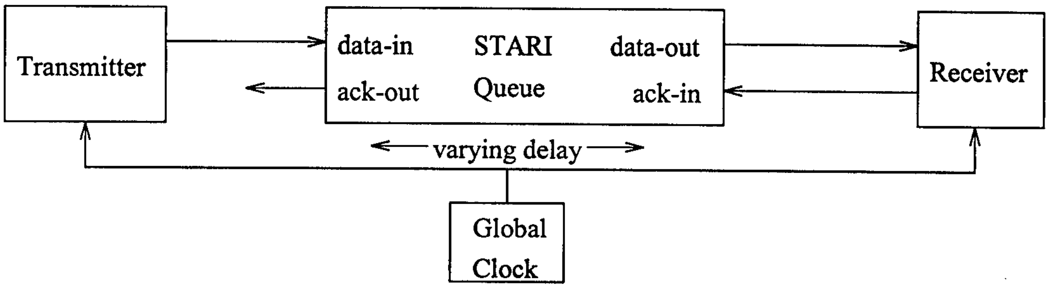


Figure 29. STARI Environment.

The general problem is to size the queue such that it buffers the data between the clock-skewed systems, allowing the transmitter to output and the receiver to input a new data value every clock cycle without waiting for the queue during "steady-state" operation (i.e., when the queue is about half full (Gre93:p35)). Note that empty is considered a data value, and that the queue is not required to output the same data it currently inputs—i.e., the queue is in fact buffering some data internally. How fast the global clock operates, how much skew is possible, and the speed of the queue components are of course the parameters that determine how many queue stages are necessary. Focusing on the queue itself avoids assumes-guarantees style verification complexity.

Figure 30 is an abstract view of the PB TSA used to formalize queue requirements. It does not explicitly show all of the states that result from the different value sequences the queue can hold. It abstracts these sequences by their length[1]. For example, the 4-element string `etef` represents

[1] During the verification, queue behavior is verified for all possible PB sequences from length $n-1$ to $n+1$.

a PB holding the sequence of four values [empty, true, empty, false], where false will be dequeued next, and empty was the last data item enqueued.

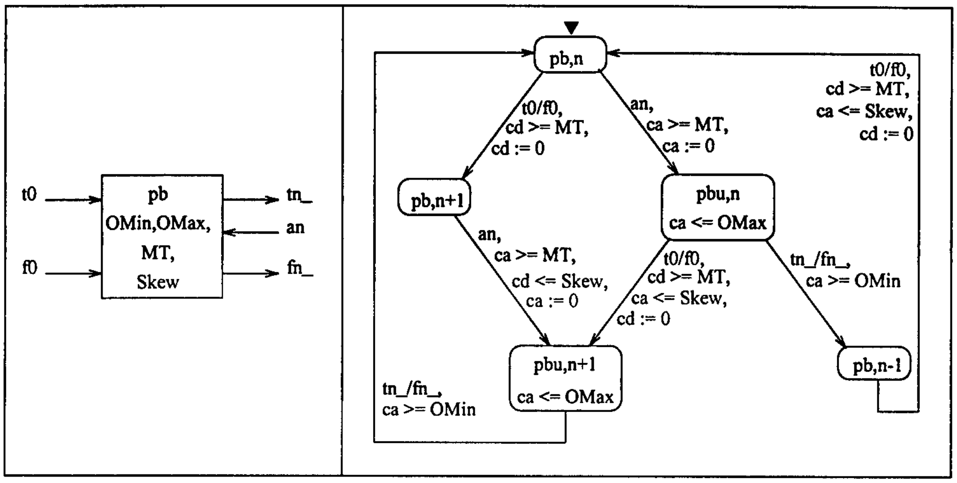


Figure 30. Perfect Buffer TSA.

PB maintains an n-size queue of information. It actually can hold either n, $n+1$, or $n-1$ (for $n > 1$) values. For $n = 2$, the three possible $(n-1)$-length values are `f`, `t`, and `e`; the n-length values are `ef`, `et`, `fe`, and `te`; $(n+1)$-length values are `efe`, `ete`, `fef`, `fet`, `tef`, and `tet`. With two n-length locations for each n-length value, one $(n-1)$-location for each $(n-1)$-value, and two $(n+1)$-locations for each $(n+1)$-value, there are $(2*4)+3+(2*6) = 23$ locations in the $n = 2$ PB TSA. Following (BM98, Gre93), it is sufficient to focus on the steady-state behavior and verify a $\lceil k/2 \rceil$-size PB against queues with k stages. Data inputs (`t0, f0`), and acknowledgment input (an) are constrained to happen at least MT time units after the previous data and acknowledgment respectively. Hence, MT is the parameter that specifies the time separation between inputs, or the inverse of the maximum frequency of the clock. The time of the last data and acknowledgment events are independently tracked by separate clocks cd and ca. After receiving an, PB will output `tn_/fn_` in the time interval [OMin,OMax] after an. OMin and OMax specify the time delay allowed for the queue to update its output after receiving an acknowledgment. Generally, OMin $\leq$ CMin, and OMax $\leq$ MT−X, where X is the sum of the Receiver's setup, hold, and acknowledgment generation delays, satisfy the PB specification.

Table 9 defines PB data input and output. Possible data input actions are determined by the values of the leftmost ($q(1)$) value in the queue in accordance with the requirements to separate true and false values by empty. Data output actions are determined by the the n^{th} ($q(n)$) and $(n-1)^{\text{st}}$ ($q(n-1)$) queue values where $q(n-1) =$ "_" is a don't care. When PB holds n values, it can either input or output a value. When it holds $n-1$ values, it can not output data, but it can input. When it holds $n+1$ values, it can not input data, but it will output. The last (right-most) value in the queue symbolizes the value that the queue is currently outputting. Input actions `t0/f0` and an are constrained to happen within Skew time units of each other.

Table 9. Perfect Buffer Input and Output.

$q(1)$	Input	$q(n-1)$	$q(n)$	Output
t	t0↓ (e)	_	t	tn_↓ (e)
f	f0↓ (e)	_	f	fn_↓ (e)
e	t0↑ (t)	t	e	tn_↑ (t)
e	f0↑ (f)	f	e	fn_↑ (f)

PB specifies that no outputs will occur when Skew input timing constraints are not met. In location [pb,$n-1$], when late inputs occur (data arrives more than Skew time units after acknowledgment), time continues to progress but no actions are possible. All legal inputs when Skew < OMin are verified following the `t0/f0` transition from location [pbu,n]. When OMin $\leq$ Skew $\leq$ OMax, data inputs from [pb,$n-1$] are disallowed for ca > Skew; these are also late inputs. Finally when Skew $\geq$ OMax, all inputs in location [pb,$n-1$] are allowed; therefore, TLCS verifies that the queue is consistent with PB for all valid time-constraint-satisfying inputs.

Since the STARI queue acknowledgment output is not connected to the transmitter, the ack_0_ of the first STARI queue stage is hidden (i.e., it becomes an internal τ(ack_0_)) to compare queue implementations to PB.

Table 10 summarizes the STARI versus PB verification results for different numbers of stages while varying Skew and maximum/minimum time (MT) separation. It shows the MT required for both the Abstract fifo-spec-based queue model (MT-A) and the Intermediate C-element-based and *Nor*-based queue model (MT-I) for TLCS verifications to fail/succeed with two to four queue stages and Skews ranging from two to six time units. OMin and OMax are tightly constrained to the same bounds as the C-element (i.e., data output must occur one to two time units after the acknowledgment input). In row 7 the "?" indicates that TLCS aborts because it ran out of global stack while computing TLC on a four-stage queue intermediate-level model for MT=8; hence there is no data for this or subsequent intermediate level TLC verifications. Verification of a gate-level queue against PB are not shown because parallel composition of a two-stage gate-level queue (eighteen gates) in TLCS aborts because of stack limitations of the current parallel composition implementation. Single-stage gate-level queue verifications succeed consistently with intermediate-level and abstract-level verifications.

Table 10. STARI$_o\underline{\prec\!\!\succ}_i$PB Results: Varying Skew & MT.

Cmin=NorMin=OMin=1, CMax=NorMax=OMax=2				
Row	Q Stages	Skew	MT-A	MT-I
1	2	2	12/none	N/A
2	3	2	8/9	6/7
3		3	8/9	6/7
4		4	8/9	7/8
5		5	8/9	8/9
6		6	9/10	9/10
7	4	2	7/8	7?
8		3	8/9	?
9		4	9/10	?
10		5	10/11	?
11		6	11/12	?
?:SWI-Prolog out of memory				

Note that for all verifications, the more abstract model is never optimistic about the results. In these verifications, the more abstract queue implementation (MT-A column) requires the same, or longer MT than the less abstract queue implementation (MT-I column) requires to model the same perfect-buffer specification. Generally, increasing the Skew while holding the number of queue stages static forces lowering the input frequency (MT increases). With Skews of four, five, and six the performance of the three-stage queue is better than the four-stage queue! This result runs counter to the intuition that longer queues will always allow more skew—it depends on how tight the timing constraints are. Note that adding an extra queue stage improves the frequency for one Skew i.e., three abstract queue stages with Skew two requires MT=9 to model PB (row 2), and with four abstract queue stages, MT=8 models PB (rows 3 and 7), but for the rest (e.g., Skew three requires MT=9 to model PB for both three-stage and four-stage queues) more queue stages do not help. This is because as queue length increases, there is a longer delay for a newly inserted datum to move through the queue, and under certain conditions, the new datum does not reach the last queue element in time to satisfy PB output constraints because OMax is too tight. This is precisely why with Skews of four, five, and six the performance of the three-stage queue is better than the four-stage queue under the tight OMax constraint.

Table 11 shows fifo-spec-level STARI verification results while holding MT=12 time units and Skew at two time units while varying OMax for two different sets of bi-bounded delays. This reports which implementations satisfy a static period and clock skew. A "Y" in columns two to four means that TLC is satisfied between the STARI queue and PB, a "N" means that TLC is not satisfied, and a "?" means that the current TLCS implementation exceeded memory limits before completing the computation. Column two reports some TLCS UltraSparc One performance statistics for the Min/Max = 1/2 verifications. These results are comparable to those in (TB97) and (BM98).

In Table 11, column two, a STARI queue implementation in steady state with gate-delays and C-element-delays in [1,2] will always accept inputs and produce outputs without delaying the transmitter or receiver as long as the inputs and outputs occur twelve or more time units after the previous input and output and within two time units of each other as is shown in (TB97). Tasiran

Table 11. STARI$_o\underline{\Diamond}_i$PB Results: Varying Imp Delays & # Stages.

Q Stages	OMax=9			OMax=10
	C & Nor Min/Max			
	1/2	Performance	2/4	2/4
1	Y	3K states 12 sec 1MB	Y	Y
2	Y	9K states 66 sec 4MB	N	Y
3	Y	37K states 7 min 16MB	Y	Y
4	Y	114K states 44 min 58MB	?	?
5	Y	452K states 4.6 hrs 254MB	?	?
6	Y	1.1M states 21 hrs 690MB	?	?
?:SWI-Prolog out of memory				

and Brayton were able to verify the behavior of an eight-stage queue with their less detailed data model, while the current TLCS implementation is limited to six stages. The TLCS verification includes checking that the sequence of correctly encoded values input to the queue are output correctly, and the TLC verification methodology needs no extra assumes-guarantees proofs.

The results in Table 11, columns three and four, are generally consistent with those in (BM98). They use a less detailed discretized intermediate-level queue model with timing delays CMin = NorMin = 2, CMax = NorMax = 4, and they claim that their STARI queue simulates the behavior of their ideal buffer for a clock period of twelve time units, and a Skew of two time units with from one to eighteen stages, and they have successfully verified up to five stages in a discretization fine enough to be equivalent to a dense model—like TLCS. Unfortunately, the TLCS results cannot be directly compared to Bozga and Maler's because the exponential state space required to handle the larger number of clocks and higher integer bounds exceeds the memory allocation limits of SWI-Prolog with more than three stages. This constrains verification to less than four stages using the more detailed dense models in TLCS. However, depending on how much set-up and hold (S&H) time is required for the receiver via timing parameter OMax, TLC fails or holds. Apparently, S&H were not factored into the STARI verifications in (BM98). If the receiver requires S&H ≤ 2, then TLCS agrees with their results. This is illustrated by the differences between columns three and four for two queue stages in Table 11. The amount of time allowed for S&H is MT - OMax. In column three, three time units are allowed for S&H, and in column four only two time units are allowed. TLC does not hold in column three when OMax = 9 (S&H = 3), and Min/Max = 2/4. The visible sequence

```
=12,an,t0,4,tn_,8,an,f0,4,fn_,8,an,f0,8,fn_,4,an,t0,8,tn_,4,an,0.001,t0,
4,tn_,7.999,an,0.001,f0,4,fn_,7.999,an,0.001,f0,7.999,fn_,4.001,f0,0.001,
an,3.999,fn_,8,f0,0.001,an,3.999,fn_,8,t0,0.001,an,3.999,tn_,8,t0,0.001,
an,7.999,tn_,4.001,an,0.001,t0,4,tn_,8,t0,1,an,3,tn_,8,f0,1,an,3,fn_,8,
f0,1,an,7,fn_,4,t0,1,an,3,tn_,8,t0,1.001,an,2.999,tn_,8,f0,1.001,an,
2.999,fn_,8,f0,1.001,an,6.999,fn_,4,t0,2,an,2,tn_,8,t0,2,an,2,tn_,8,f0,2,
an,2,fn_,10,an,1,f0,4,fn_,7,an,1,f0,4,fn_,7,an,1.001,f0,4,fn_,6.999,an,
1.001,f0,7.999==>
```

leads to the states and combined time vector:

```
 I:[fifo_specFF,fifo_specFEU]
 S:[pbu,fe]
 T:[[[k0,3,4],[k1,3,4],[k2,9,9],[k3,7,8]],[[k0,k1],[k3],[k2]]]
 Where possible Imp Actions are: [f0,fn_]
 and possible specification Actions are:  [fn_]
and future Imp outputs are not matched by the Spec.

 IInv:[[k1,leq,4]]
 A:f0 G:[] R:[] I_:[fifo_specEF,fifo_specFEU]
 A:fn_ G:[[k1,geq,2]] R:[k1] I_:[fifo_specFF,fifo_specFFU]

 SInv:[[k2,leq,9]]
 A:fn_ G:[[k2,geq,2]] R:[] S_:[pb,f]
```

where the implementation's `fn_` output can occur later than the specification's `fn_` output because the specification cannot stay in location [pbu,fe] after clock k2 = 9. This late output violates the TLC relation because it infringes on the S&H requirements of the receiver, but TLC holds in column four when OMax = 10 (i.e., receiver S&H is tightened from three to two time units). In this case, if S&H requirements of the receiver are less than or equal to two time units, then TLC holds and TLCS confirms Bozga and Maler's results. Unfortunately, the current TLCS stack limitations keep us from checking their results for more than a three-stage queue when using queue-stage models with bi-bounded delays [2,4].

The original proof of correctness for STARI is found in Greenstreet's dissertation. Comparing his model and its limitations to the TLCS verification enlightening. First, focus on the similarities. The most important similarity is that TLCS and Greenstreet both model STARI with discrete functional events rather than the analog voltage levels (Gre93:p146). Second, neither TLCS nor Greenstreet prove properties about the initialization of the queue. Both "proofs" start by assuming an initialized queue that is half full. Greenstreet argues convincingly that initialized conditions are easy to establish in the implementation so it is not an issue for either "proof."

In Greenstreet's original STARI proof of correctness, he focused on verifying that the signaling protocol of the queue was observed by formulating and proving invariant properties about the queue's timing expressed as synchronized transitions, but he did not prove some other correctness criteria are satisfied (Gre93:p144), and here is where there are some significant differences between the "proofs." For example, he did not prove that the sequence of values output by the FIFO are the same as those input because he did not model the actual data values themselves. In contrast, the TLCS STARI verification does model the data values explicitly, and since the perfect-buffer outputs the sequence of values it receives, TLCS verifies that the queue does as well. Greenstreet's detailed proof also models the transmitter and receiver repeatedly synchronizing exactly at specified time intervals; since the TLCS verification explicitly allows up to Skew time units between the data and acknowledgment inputs, the TLCS verification is more general in that sense. Perhaps the most significant limitation of his analysis is that his FIFO stage models abstract the data and acknowledgment outputs into a single atomic action (Gre93:p145). Since these are actually three separate signals produced by three components with their own distinct delays in his implementation, the TLCS model is more realistic and therefore closer to verifying the actual circuit behavior. Unfortunately, his abstractions prevent thorough quantitative comparison of his results directly with TLCS's, but a single counterexample suffices to show the need to model and "prove" the circuit's behavior with higher fidelity models. Greenstreet derives Formula 34 (Gre93:p33) for the skew tolerance (Λ) of an n-length queue with period π and C-element delay δ.

$$\Lambda = (n+1)(\pi - 2\delta) \tag{34}$$

According to Table 11, column 3, where $n = 2 =$ Q Stages, $\pi = 12 =$ MT, and $\delta = 4 =$ CMax, Greenstreet says $\Lambda = 12$, but TLCS produces the late-output counterexample with only a skew of two as discussed above. Even if $\pi = 9 =$ OMax, $\Lambda = 3 > 2$; this remains a late-output counterexample. Fundamentally, Greenstreet's results differ from the TLCS results because the delay of the *Nor* used to compute the acknowledgment signals in the implementation is ignored by his analysis when he assumes that each C-element computes its data output and acknowledgment simultaneously.

6.2.5 Comparing Verification Methodologies. Generally, from gate-level to Perfect Buffer, the TLC verification methodology requires three verifications to hierarchically verify the abstract Perfect buffer against a gate-level implementation. At the top-most level, a monolithic $\lceil n/2 \rceil$-size PB is the specification (Figure 30), and the implementation is a n-stage STARI FIFO Queue composed from n Queue Stage TSA (Figure 28); i.e., the verification proves $QS_1 || QS_2 || \ldots || QS_n \mathrel{{}_{o}\underline{\diamond}_{i}} PB(n/2)$. The second verification specification is a single STARI FIFO Queue Stage specification, and the implementation is a three-component parallel-composed C-element-*Nor* intermediate-level queue stage (Figure 26); i.e., the verification proves $C_1 || C_2 || Nor \mathrel{{}_{o}\underline{\diamond}_{i}} QS$. The final verification specification is the Wobbly-C-element specification (Figure 25) against its four-component *And-Or* gate-level implementation (Figure 4); i.e., the verification proves $And_1 || And_2 || And_3 || Or \mathrel{{}_{o}\underline{\diamond}_{i}} C$. In the preceding discussion, results from several other verifications are presented to show that composing components and verifying across more than one level of abstraction are easy using TLCS.

In contrast, applying the assumes-guarantees verification methodology to STARI (TAKB96, TB97) is more expensive and is not as detailed. It requires more verifications and the construction of extra abstract models, and it does not verify a gate-level implementation of the C-element.

Most of the extra verifications are required to show that the abstract FIFO stage model depicted earlier in Figure 27 is a correct abstraction of the Figure 26 intermediate level queue stage in its environment. Let F and A denote the intermediate level queue stage and abstract FIFO stage timed processes respectively, and Tx and Rx denote the transmitter and receiver timed processes modeling the queue environment. Proving the abstraction is valid requires proving for an n-length queue that

$$Tx||F_1||F_2||\ldots F_n||Rx \preceq_L Tx||A_1||A_2||\ldots A_n||Rx$$

Since the environment for each pair of models (e.g., F_2, A_2) is different, n separate assumes-guarantees proofs are normally required. They were able to construct models E_{left} and E_{right} generalizing the environment to the left and right of the i^{th} component and reduce the number of verifications required. This reduces the main verification required to

$$E_{\text{left}}||F||E_{\text{right}} \preceq_L E_{\text{left}}||A||E_{\text{right}}$$

at the expense of showing that E_{left} and E_{right} are correct abstractions for the left and right-hand sides of each module i; i.e.,

$$\forall_i \in [1..n][Tx||A_1||A_2||\ldots A_{i-1} \preceq_L E_{\text{left}} \wedge A_{i+1}||A_{i+2}||\ldots A_n||Rx \preceq_L E_{\text{right}}]$$

This can be shown by induction on i by

1. showing $Tx \preceq_L E_{\text{left}}$[2].

2. Assuming $Tx||A_1||A_2||\ldots A_{i-1} \preceq_L E_{\text{left}}$ and showing $Tx||A_1||A_2||\ldots A_i \preceq_L E_{\text{left}}$.

3. Concluding $E_{\text{left}}||A_i \preceq_L E_{\text{left}}$.

So, the assumes-guarantees methodology requires six verifications:

1. $Tx \preceq_L E_{\text{left}}$

[2] This requires disallowing the transmitter to change its data output if the first stage has not copied the transmitter's previous output value (caring about *ack_0_*) and later proving that the transmitter never wants to modify its data output while the first stage is not ready.

2. $E_{\text{left}} || A \preceq_L E_{\text{left}}$

3. $Rx \preceq_L E_{\text{right}}$

4. $A || E_{\text{right}} \preceq_L E_{\text{right}}$

5. $E_{\text{left}} || F || E_{\text{right}} \preceq_L E_{\text{left}} || A || E_{\text{right}}$

6. $Tx || A_1 || A_2 || \ldots A_8 || Rx$ satisfies the two timing properties enumerated earlier:

   (a) "Each data value output by the transmitter must be inserted into the FIFO before the next one is output." (i.e., the transmitter must not change input values to the queue until the first FIFO stage acknowledges receiving the input by generating an ack_0_ event.)

   (b) "A new value must be output by the FIFO before acknowledgment from the receiver." (i.e., the receiver must not generate an ack_n_ event to the FIFO until it has received the data from the last FIFO stage.)

and four abstract models (Tx, Rx, E_{left}, E_{right}) that the TLC methodology does not require. For each of the first five verifications, COSPAN requires inputting untimed mappings between the state spaces of the models being compared. TLCS' single C-element/*Nor* versus Queue-Stage specification verification is equivalent to these five verifications and does not require the higher-order inductive logic step.

The sixth verification is required to prove properties that are verified directly by the TLC verification between PB and the Queue. In the first case, PB's input of a value must always be matched by Formula 22; in the second case, all outputs produced by the Queue must be allowed by PB according to Formula 25. The authors did not explain the property verification (number 6), but it must be a model checking proof; interestingly, it required over 37 times the amount of time and 14 times the memory of the other proofs. In fact, they were unable to complete this verification for a 3-stage F-model FIFO using 1GB of memory (TB97). As shown in Table 10 TLCS completes verification on similar complexity 3-stage models (the MT-I column, rows 2-6).

Although the models used in Bozga and Maler's STARI verification are described in detail, the verification methodology is not described in detail (BM98). The implementation models they use are at an intermediate level of complexity and do not go all the way down to a gate-level C-element implementation. Their "ideal buffer" specification model of the queue in its environment is similar to perfect buffer, except that it consists of four timed automata: clock, transmitter, receiver, and queue. They use a Binary Decision Diagram (BDD) extension of the Kronos tool (ABK$^+$97) to compute that n-length intermediate-level queues simulate ideal buffers of size n when both are initialized with $n/2$ data values for $n \in [1..18]$. They use a discrete model of time with integral time steps. They claim that the discrete semantics coincide when they use $1/k$ time steps for k-clock systems, and with that fine a discretization, they are only able to verify $(n = 5)$ stage models compared to the $(n = 3)$ stage TLCS verification.

Since the Kronos verification methodology is based on the simulation relation with models of the environment, the methodology is also bound by the assumes-guarantees rule. The discretization simplification allows them to perform the whole verification at once saving the effort required to verify the decomposition of the verification. They do not need to do the 5 extra verifications required with the COSPAN methodology or build the models E_{left} and E_{right} when their tool handles the state space explosion of the entire verification. Whenever the entire verification cannot be done at once, they too must construct extra abstract models and do the associated verifications to check their validity. In any case, they build three models: clock, transmitter, and receiver, to support their verification methodology. These models are not required for the TLC verification.

6.3 Summary

This chapter demonstrated the utility of Timed Safety Automata models and the Timed Logic Conformance (TLC) relationship for systems engineering and verification. It described canonical system models for monotonic and inertial hardware primitives and explained the results of TLC verifications at several different levels of abstraction, and it compares the TLCS results and verification methodology with other published work.

Generally, despite modeling systems in more detail than others, TLCS is able to compute comparable results despite the fact that it explicitly enumerates the states and is written in Prolog. Using the asynchronous STARI verification problem as a benchmark, TLCS confirms Berkeley researchers results (TB97); the TLCS models extend the verification to include data values passing correctly through the queue and no assumes-guarantees reasoning is required to accomplish the verification. Comparing the TLCS results with French researchers from VERIMAG (BM98), TLCS generally confirmed their results but pointed out an important counterexample when set-up and hold time requirements of the receiver are taken into account. The TLCS model is much more detailed than the original proof of STARI correctness (Gre93), proving properties about the actual data transferred as well as showing a counterexample to the formula derived for allowable skew between sender and receiver clocks when the more realistic model is used.

TSA are well suited for modeling systems at various levels of abstraction, and the TLC relationship is useful for verifying when one TSA is an acceptable implementation of another. TSA modeling and TLC verification support incorporating the environmental constraints into specifications in a natural way. This reduces the modeling problem by eliminating environmental models, and the incorporated environmental constraints minimize the number of states that must be examined, making a fair tradeoff possible between model fidelity and computational complexity.

The TLC verification methodology is simpler than the assumes-guarantees methodology because no assumes-guarantees proof obligations or extra abstract models are required to support decomposing the verification task.

This chapter's contributions are:

- Definition and application of canonical inertial and monotonic hardware modeling techniques.
- Demonstration of a simple and relatively efficient verification methodology that supports using more detailed models and discovers subtle problems not exposed by others.
- A comparison and critique of TLCS verification results against other published work.
- A comparison and critique of the TLC verification methodology against other published work.

VII. Conclusions

This chapter summarizes the problem and lists the research objectives. Then, it enumerates and explains the research contributions. After proposing some future research opportunities, it concludes with final remarks.

7.1 Summary

Chapter II defined and described several example formalisms for modeling and reasoning about the "equivalent" behavior of concurrent systems. There are some significant expressiveness problems with these formalisms. Upper and lower time bounds (bi-bounded delays) are difficult to define in the simplest models. Some of the simpler models, and even the more complicated ones, have nonintuitive semantics such as the maximal-progress semantic leap from two processes waiting individually to perform their actions to cooperating processes that cannot wait to perform their cooperative actions. None of the process-algebra-style models support expressing general temporal relationships between actions that do not sequentially follow each other. Timed processes support expressing general temporal relationships between actions, but they are quite complicated because they use state functions to define outputs and invariants, and sequences of events to define process semantics. Furthermore, in order for timed processes to be reasoned about consistently, they must be nonblocking. This dramatically increases the complexity of both the model building and verification task.

Chapter II also discussed the timed "equivalence" relationships Timed Bisimulation and Weak Timed Bisimulation for TCCS agents, CTR Refinement for CTR agents, and Timed Simulation and Timed Implementation between timed processes. The bisimulation relationships generally restrict designer freedom too much and do not allow efficient implementations. The CTR refinement relationship is looser, but CTR allows implementations that do not accept all the inputs accepted by the specification, so CTR is formalized "backwards." Timed Simulation is better than the bisimulation straight jacket, but its assumes-guarantees methodology requires many iterations of expensive verifications just to "verify the verification" because of circular dependencies between environment and system models. Chapter II revealed that the existing tools for the most powerful methodology require a lot of user input that is not straight forward to supply.

In summary, Chapter II demonstrated the need for a simpler and yet powerful modeling formalism to accurately express the relationship between behavior and time. Designers also need a more practical mathematical relationship between models that supports an automated verification methodology that factors in environmental timing properties without building many different models of the environment and using them to "verify the verification."

These were the specific research objectives:

1. Adopt or create a simple modeling formalism rich enough to express discrete-valued behavioral properties and timeliness requirements of digital circuits while modeling continuous time.

2. Canonically define how to model digital circuit components and specify required behaviors and timing using the modeling formalism.

3. Formally define a *practical* relationship that expresses when one model satisfies the timing and behavioral requirements of another. Prove that the relation has the necessary mathematic properties for meaningful verification.

4. Write a tractable computational procedure that calculates when the relation holds between two models.

5. Demonstrate the utility of the relation on benchmark digital circuit design problems.

6. Define a verification methodology for using the relation to efficiently hierarchically verify larger systems.

These objectives have been accomplished.

7.2 Contributions

After enumerating contributions organized by topic, the following sections summarize and explain them.

- TSA Model of Computation

    1. Simpler than timed processes.
    2. More expressive than most other "simple" models. Theoretically unlimited power to express the timing relationships between actions.
    3. A formal definition of synchronous parallel composition, and a useful implementation of the composition procedure in TLCS.
    4. Simple rules and canonical forms for modeling hardware components as TSA.

- TLC Formal Relationship Between Models

    1. Safely weakens "equivalence" and gives designers the structural, temporal, and behavioral freedom they need to design and reuse efficiently.
    2. Relaxes timing requirements "the right way."
    3. Relatively efficient—avoids checking irrelevant state space.
    4. "Completes" the model checking verification process.
    5. An "efficient" implementation of the TLC decision procedure and demonstration of its utility on benchmark problems.

- Verification Methodology

    1. A more powerful and efficient hierarchical verification methodology.
    2. Breaks the verification task down into independent sub-verifications.

3. Avoids always considering changes in the environment and other modules at every level of design.
4. Fewer abstract models required.
5. Requires no user-supplied state relation.

7.2.1 Model of Computation. The "simple" and expressive Timed Safety Automata (TSA) model of computation as adapted for this research does not suffer the deficiencies revealed in Chapter II. TSA support high-fidelity modeling of electronic circuits when constrained as required by the unique Def. 30 modeling constraints and bi-bounded inertial delay modeling techniques. The formal synchronous TSA parallel composition rules and TLCS implementation of them support modeling and equivalence checking large and complex circuits.

TSA suffer none of the expressiveness problems associated with untimed process algebras. Upper and lower time bounds (bi-bounded delays) are easily defined using TSA location invariants and transition guards. The maximal-progress semantic leap (from two processes waiting individually to perform their actions to cooperating processes that can not wait to perform their cooperative actions) does not exist in the TSA parallel composition rules. General temporal relationships between actions that do not sequentially follow each other are easy to express in TSA by resetting a clock and freely using clock predicates to define the relationship.

The Mealy machine TSA model is simpler than the Moore machine COSPAN timed process model. TSA do not define output by associating functions with locations. The TSA model is easier to use. Users need not specify behaviors for all inputs in all states for all possible times at every level of the hierarchy. Only those inputs necessary to satisfy the TLC relation with the specification and satisfy the CI-free property in compositions must be defined. In another sense, the TSA model is more expressive than both timed processes and process algebras because TSA allow users to use $\{\leq, \geq, <, >\}$ instead of just $\{\leq, \geq, =\}$ to define timed behavior using clock constraints. The TSA model is also simpler than the most expressive process-algebraic model because it requires only about half of the rules to define model semantics and parallel composition.

The TSA model works well for describing behavior at many different levels of abstraction. Chapters IV and VI specified novel rational modeling constraints and defined canonical ways to monotonically and inertially model implementation-level primitive hardware circuits. Chapter VI demonstrated using those models in example verifications from handfuls of gates to a dual-rail-encoded queue for interfacing systems with clock skew between them.

7.2.2 Formal Relationship Between Models. Chapter IV formally defined the timed equivalence relation, Weak Timed Bisimulation, that relates Dense Labeled Transition Systems (DLTS's) with different internal action sequences but the same observable action sequences and timing. To relate systems that do not have the exact same timing, Chapter IV also defined how to abstract away the temporal differences between TSA, and how to use those abstractions to weaken Weak Timed Bisimulation via the partial order Timed Logic Conformance (TLC) relation.

With a few well-defined exceptions, TLC requires that implementation inputs are a timed superset of specification inputs and that implementation outputs are a timed subset of specification outputs. TLC formalizes these notions and specifies when an implementation can safely replace a specification, and it has the necessary mathematical properties to support hierarchical verification of larger systems with the exception that one must be careful when the most abstract specification is parallel composed.

In summary, TLC:

- Pragmatically and intuitively weakens "equivalence" and gives designers the freedom they need to design and reuse designs efficiently. TLC provides greater structural, temporal, and behavioral freedom of implementation while maintaining a meaningful and accurate implementation "implements" specification relationship.

- Relaxes timing requirements "the right way." Instead of accepting implementations that can refuse specification inputs, TLC rejects them; at the same time it rejects implementations that output when the specification does not allow outputs.

- Avoids checking irrelevant implementation state space by ignoring extra input derivatives that the specification does not have.

- Is a "completing" companion to model checking. Since TLC gives designers substantial freedom of design, it does not generally preserve arbitrary timed modal logic or μ-calculus properties, yet all pragmatic properties are preserved when the specification completely defines the inputting environment for the implementation.

Further, Chapter V described the reasonably efficient region-automata-based decision procedure implemented in TLCS. TLCS computes whether or not the TLC relation holds with a minimum of user input required. TLCS computed whether or not TLC holds for several examples including the STARI (Self-Timed at Receiver's Input) asynchronous circuit for communicating safely between two clock-skewed systems. The results, summarized in Chapter VI, are comparable to those published elsewhere (Gre93, BM98, TB97).

7.2.3 Verification Methodology. The powerful and relatively efficient top-down TLC hierarchical verification methodology also works bottom-up. The TLC verification methodology is better than assumes-guarantees reasoning because it simplifies and reduces the burden of building models, and it breaks the verification down into less complex and independent pieces.

The TLC verification methodology is simpler because it can be independently decomposed without the assumes-guarantee circular dependency verifications. This reduces the magnitude of the verification task tremendously because iteratively changing models and specifications only affects the verifications up and down the hierarchy, not across the breadth of it for every iteration.

In summary, TLC verification:

- Breaks the verification task down into independent sub-verifications that are smaller and more tractable.
- Avoids having to always consider changes in the environment and other modules at every level of design.
- Requires no environmental model: naturally captures environmental timing requirements in the top-level specification.
- Provides an efficient alternative to assumes-guarantees proof obligations.
- Requires no user-supplied state relation.

7.3 Future Work

There is always more work to be done. The plans and ideas are organized into three areas; TLCS enhancements, TSA/TLC theory extensions, and promising TLC applications.

7.3.1 TLCS Enhancements. SWI-Prolog indexes information on the heap by the first term of the asserted fact by default, but facts can be indexed by more non-list terms to improve efficiency. It would greatly speed up the TLC process if the list of visited state pairs and their time vectors could be directly indexed according to all three terms—i.e., I's location, S's location, and the time vector. Unfortunately, the time vector data structure is a list and cannot be used as a hash key. Generating a nearly unique atomic hash-key for each time vector and using it to index visited states and to represent time vectors on the stack seems feasible and would vastly improve the performance of the SWI-Prolog TLCS.

The current parallel composition algorithm stores the transition relations of the composed systems on the stack and recursively calls itself until no new states are reached. For small compositions, this approach works fine, but for large compositions (e.g., the two-queue-stage STARI queue with 18 subcomponents) TLCS overflows SWI-Prolog's stack limitation (64MB). Updating the parallel composition algorithm to use the heap (up to 1.9GB) instead of the stack to store component transition sequences would improve reasoning about larger systems.

There have been advances that reduce the exponential space complexity of the region automata time representation (HKWT95, LLPY97). These techniques should be studied to see if TLCS space complexity can be significantly improved. Many techniques are applicable for model checking because they depend on the formulas being checked and the applicable state space to minimize the complexity. Some techniques have been applied to equivalence checking; e.g., clock minimization algorithms, clock-planes, and geometric clock regions (RM94). Generally the efficiency gains depend on the particular relationships between the clock resets and clock predicates in the specific TSA. Whether these techniques can be directly applied to greatly improve the TLCS efficiency or if they would require extensions or modifications to the theory or algorithm itself is not yet clear. Of particular interest is University of Utah's Timed Event Level (TEL) structure research (BM97, BMH99). TEL structures are efficient ways of expressing timed Petri-net style behavior and signal level information together. University of Utah's ATACS system uses geometric representations of clock regions to efficiently reason about TEL structure timed state spaces representing the environment and the system. Conflicts between the environment and system state spaces are timing failures and design errors.

7.3.2 TSA/TLC Theory Extensions. One promising extension of TLC theory is to define a "confluent" Timed Logic Conformance. i.e., weaken TLC such that implementations can satisfy confluent specification output bursts with any of the allowed sequences of outputs. Frequently, specifications allow outputs to be generated in any sequence, but the output sequences converge to a single state. Currently, the TLC relation requires the implementation to generate all of the specification's output sequences. Since considerably more efficient implementations can typically be made that produce only a subset of those sequences, and the receiver of that sequence typically does not care which order they arrive, a confluent TLC relation would safely give designers more behavioral freedom.

A second extension is to develop a methodology for safely abstracting actions/events from binary values changing to events on groups of binary signals, events on numbers, strings, and records. While not theoretically a problem, the rapid growth of region-automata timed state space severely limits reasoning about the behavior of multi-bit hardware architectures with TLCS. Current plans are to scale down the complexity by reducing bit-width, but more abstraction options are needed—especially if TLCS is used for system-level hardware and software architectures.

7.3.3 Promising TLC Applications. Integrating a TLC decision procedure and existing tools already available for different applications should be investigated. A main problem with the application of formal methods to real design is that there are many different models of computation and tools to support them but no Rosetta stone or transformation process for most of the models. Instead of trying to promote the use of TLCS with this flavor of TSA formalism for equivalence checking only, defining the TLC relation for an expressive timed FSM formalism already used for model checking or other formal methods application makes sense. One model checking environment without an efficient equivalence checker is the Concurrency Factory (CGL$^+$94). Integrating TLC into the Concurrency Factory would package the TLC theory in a useful way and expose more people to the more efficient TLC verification methodology.

Using TLC to define semantics of system architecture refinement in a category-theory-based specification-refinement tool like SpecWare(SJ94) should be investigated. Mark Gerken laid the foundation for using untimed process algebras to formally define different software architectures and reason about them (Ger95). Since the timing of hardware and software systems working together is so critical to their function, extending his theory over a timed FSM formalism like TSA and using the TLC relation to define an appropriate implementation relation between system architectures makes sense.

Another potentially fruitful research area is defining mappings from the Unified Modeling Language (UML) (Dou98) into TSA for the purpose of defining and proving temporal claims about UML specifications, architectures, and implementations. UML is a popular language for specifying system architectures, but UML does not have a solid theoretical foundation for all of its semantics. Previous AFIT research successfully defined formal semantics for informal graphically-based object-oriented specification and modeling languages similar to UML (DeL96), but the formal semantics have not been extended to the specification and reasoning about system timing and the relationship of time and behavior. Since one of the most widely used views of a UML specification is FSM-based, using the TSA model and TLC relation to formally define the relationship between time and behavior and to refine the FSM part of UML specifications makes sense.

Another very important research idea is to use temporal logic, abstract TSA models, and the TLC relation to define and warrant the behavior of intellectual property (IP) subsystems. If the interfaces to the subsystem are defined as TSA, and the actions of the subsystem are defined using temporal logic or timed modal μ-calculus formulae, then potential users of the subsystem could compose the subsystem into their application and determine if they can interface with the subsystem correctly and satisfy their performance specifications. A predicate-logic based specification of behavior would likely be necessary to describe the data-path function of the subsystem as done at the University of Cincinnati (Bar98). This specification would be used for reasoning about the system's functional consistency and correctness with a theorem prover instead of a model checker or equivalence checker. With such formal specifications for IP components, users could automatically and reliably search for and reason about the suitability of the IP for their application without forcing the IP vendor to compromise the details of their implementation.

7.4 Concluding Remarks

Based on the results and conclusions in this and previous chapters, the research objectives have been successfully achieved. The Timed Safety Automata (TSA) formalism is rich enough to express hardware behavioral properties and all the necessary timeliness requirements. How to use TSA to model hardware components and specify required behaviors and timing was canonically defined. An efficient parallel TSA composition procedure was defined to support design and verification of more complex systems. The Timed Logic Conformance (TLC) relation was formally defined and specifies when one TSA satisfies the timing and behavioral requirements of another TSA. TLC is loose enough to give designers the structural, temporal, and behavioral freedom they need to implement efficiently. TLC does not sacrifice the fundamental requirements to match specification inputs and safely allow implementation outputs. The TLC partial order has the necessary mathematic properties for meaningful verification. The tractable computational procedure (TLCS) calculates when the TLC relation holds. TLCS successfully demonstrated the utility of the TLC relation on benchmark circuit design problems, and supported the development of a powerful and relatively efficient top-down hierarchical verification methodology.

Bibliography

ABK^{+}97. E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data structures for the verification of timed automata. In O. Maler, editor, *Proc. HART'97*, LNCS 1201. Springer, 1997.

ACD90. Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proceedings of 5th IEEE Symposium on Logic In Computer Science*, pages 414–425, 1990.

ACD93. Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

ACH94. R. Alur, C. Courcoubetis, and T. Henzinger. The observational power of clocks. In *Proceedings of CONCUR '94*. LNCS 836, 1994.

AD94. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

All83. James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

All84. James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, July 1984.

ANB95. John Aldwinckle, Rajagopal Nagarajan, and Graham Birtwistle. *An Introduction to Modal Logic and its Applications on the Concurrency Workbench.* Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, February 1995. (Preliminary Version).

AR91. V.S. Alagar and G. Ramanathan. Functional specification and proof of correctness for time dependent behavior of reactive systems. *Formal Aspects of Computing*, 03(3):253–283, Jul-Sep 1991.

Bar98. Phillip Baraona. *The Syntax and Semantics of VSPEC, a Larch/VHDL Interface Specification Language.* PhD thesis, University of Cincinnati, 1998.

BGS89. Thomas Bihari, Prabha Gopinath, and Karsten Schwan. Object-oriented design of real-time software. In *Proceedings of the Real-Time Systems Symposium*, pages 194–201, LosAlamitos, CA, 1989. IEEE Computer Society Press.

BM97. Wendy Belluomini and Chris J. Myers. Timed event/level structures. In *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 199–208. ACM/IEEE, December 1997.

BM98. Marius Bozga and Oded Maler. Modeling and verification of the STARI chip using timed automata. In *Proceedings of Cav '98*, 1998.

BMH99. Wendy Belluomini, Chris J. Myers, and H. Peter Hofstee. Verification of delayed reset domino circuits using ATACS. In *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 39–44. ACM/IEEE, March 1999.

BS91. J.A. Brzozowski and C-J. H. Seger. Advances in asynchronous circuit theory Part II: Bounded inertial delay model, MOS circuits, design techniques. EACTS Bulletin 43, 1991.

Bur92. Jerry Burch. Delay models for verifying speed-independent asynchronous circuits. In *Proceedings of the International Conference of Computer Design (ICCD)*, pages 270–274. IEEE Computer society Press, October 1992.

Cer92. K. Cerāns. Decidability of bisimulation equivalences for parallel timer processes. In *Proceedings of CAV '92*. LNCS 663, 1992.

Cer95. Kārlis Cerāns. CTR: A calculus of timed refinement. In I. Lee and S. Smolka, editors, *Proceedings of CONCUR '95*, pages 516–630, 1995.

CGL93. Kārlis Cerāns, Jens Chr. Godskesen, and Kim G. Larsen. Timed modal specification-theory and tools. In C. Courcoubetis, editor, *Proceedings of CAV '93*, pages 253–267, 1993.

CGL$^+$94. J. N. Cleaveland, J.N. Gada, P. M. Lewis, S. A. Smolka, O. Sokolsky, and S. Zhang. The concurrency factory—practical tools for specification, simulation, verification, and implementation of concurrent systems. In *Proceedings of The DIMACS Workshop on Specification Techniques for Concurrent Systems, Princeton, NJ*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1994.

CH92. P.J. Clarke and D.J. Holding. The specification, design and verification of real-time embedded control logic using csp and tcsp. In *Real-Time Programming (WRTP '92). Preprints of the IFAC Workshop*, pages 167–172, Oxford, UK, June 1992. Pergamon Press.

CHLM93. P.C. Clements, C.L. Heitmeyer, B.G. Labaw, and A.K. Mok. Applying formal methods to an embedded real-time avionics system. In *Proceedings of the First IEEE Workshop on Real-Time Applications*, pages 46–49, Los Alamitos, CA, May 1993. IEEE.

CL96a. Duncan Clarke and Insup Lee. Automatic specification-based testing of real-time properties. ftp.cis.upenn.edu/pub/rtg/Paper/Full_Postscript/test96.ps.Z, 1996.

CL96b. Duncan Clarke and Insup Lee. A hybrid approach to formal verification applied to and atm switching system. Technical Report MS-CIS-96-04, University of Pennsylvania, 1996. ftp.cis.upenn.edu/pub/rtg/Paper/Full_Postscript/atm.ps.Z.

CLK94. Jin-Young Choi, Insup Lee, and Inhye Kang. Timing analysis of superscalar processor programs using acsr. In *Proceedings of 11th IEEE Workshop on Real-Time Operating Systems and Software*, 1994.

CW96. Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178, Carnegie Mellon University, August 1996.

Dan92. Mats Daniels. Modelling real-time behavior with an interval time calculus. In *Formal Techniques in Real-Time and Fault-Tolerant Systems. Second International Symposium Proceedings*, 1992.

Das85. B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, 11(1):80–86, January 1985.

Dav93. Jim Davies. *Specification and Proof in Real-Time CSP.* Cambridge University Press, 1993.

DeL96. Scott A. DeLoach. *Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specifications.* PhD thesis, Air Force Institute of Technology, 1996.

Dou98. Bruce Powel Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems.* Grady Booch, Ivar Jacobson, and James Rumbaugh Object Technology Series. Addison-Wesley, 1998.

Dro89. Geoff Dromey. *Program Derivation: The Development of Programs from Specifications.* Addison-Wesley, Sydney, Australia, 1989.

EAP98. O. Maler E. Asarin and A. Pnueli. On discretization of delays in timed automata and digital circuits. In R. de Simone and D. Sangiorgi, editors, *Proc. Concur'98*, LNCS. Springer, September 1998. to appear.

FH92. Stephen Fickas and B. Robert Helm. Knowledge representation and reasoning in the design of composite systems. *IEEE Transactions on Software Engineering*, 18(6):470–482, June 1992.

FP93. Laurent Fribourg and Marcos Veloso Peixoto. Concurrent constraint automata. Technical Report LIENS 93-10, Ecole Normale Superieure, ftp.ens.fr/pub/reports/liens/liens-93-10.A4.ps.Z, 1993.

Ger95. Mark James Gerken. *Formal Foundations for the Specification of Software Architecture.* PhD thesis, Air Force institute of Technology, 1995.

GF88. A. Gabrielian and M. K. Franklin. State-based specification of complex real-time systems. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 2–11, Huntsville, AL, 1988. IEEE Computer Society Press.

GF90. Armen Gabrielian and Matthew K. Franklin. Multi-level specification and verification of real-time software. In *Proceedings of the 12th International Conference on Software Engineering*, pages 52–62. IEEE Computer Society Press, March 1990.

GI91. A. Gabrielian and R. Iyer. Integrating automata and temporal logic: A framework for specification of real-time systems and software. In C.M.I. Rattray and R.G. Clarke, editors, *Proceedings of the Unified Computation Laboratory.* Oxford University Press, 1991.

GI92. A. Gabrielian and R. Iyer. Verifying properties of hms machine specifications of real-time systems. In *Proceedings of the Workshop on Computer-Aided Verification— Aalborg, Denmark, July 1-4, 1991, Lecture Notes in Computer Science.* Springer-Verlag, 1992.

Gre93. Michael R. Greenstreet. *STARI: A Technique for High-Bandwidth Communication.* PhD thesis, Princeton, January 1993.

GSSAL94. Rainer Gawlick, Roberto Segala, Jørgen Søgaard-Andersen, and Nancy Lynch. Liveness in timed and untimed systems. In *Proceedings of 21st ICALP*, LNCS 820, pages 166–177. Springer-Verlag, 1994.

GV95. Daniel Gajski and Frank Vahid. Specification and design of embedded hardware-software systems. *IEEE Design and Test of Computers*, Spring 1995.

Hal92. W. A. Halang. Real-time systems: Another perspective. In Krishna M. Kavi, editor, *Real-Time Systems Abstractions, Languages, and Design Methodologies*, pages 11–18. IEEE Computer Society Press, 1992.

HB94. Henrik Hulgaaard and Stephen M. Burns. Bounded delay timing analysis of a class of CSP programs with choice. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11. IEEE, November 1994.

Hen95. Thomas A. Henzinger. Hybrid automata with finite bisimulations. In F. Fulop, Z.; Gecseg, editor, *Automata, Languages and Programming. 22nd International Colloquium, ICALP 95 Proceedings*, pages 324–335. Springer-Verlag, 1995.

HJ95. Michael G. Hinchey and Stephen A. Jarvis. *Concurrent Systems: Formal Development in CSP.* McGraw-Hill International Series in Software Engineering. McGraw-Hill Book Company Europe, London, 1995.

HKWT95. Thomas A. Henzinger, Peter W. Kopke, and Howard Wong-Toi. The expressive power of clocks. In F. Fulop, Z.; Gecseg, editor, *Automata, Languages and Programming. 22nd International Colloquium, ICALP 95 Proceedings*, pages 417–428. Springer-Verlag, 1995.

HLY91. Uno Holmer, Kim Larsen, and Wang Yi. Deciding properties of regular real timed processes. In *Proceedings of CAV '91*, pages 443–453, 1991.

HNSY94. Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

Hoa85. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice/Hall International, UK, LTD., 1985.

HP87. Derek J. Hatley and Imatiaz A. Pirbhai. *Strategies For Real-Time System Specification.* Dorset House Publishing, New York, 1987.

Jah89. Farnam Jahanian. Verifying properties of systems with variable timing constraints. In *Proceedings, Real Time Systems Symposium(Cat No.89CH2803-5)*, pages 319–328, New York, December 1989. IEEE Computer Society Press.

Jen93. E. Douglass Jensen. A timeliness model for asynchronous decentralized computer systems. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*, pages 173–182, Los Alamitos, CA, 1993. IEEE Computer Society Press.

JM86. Farnam Jahanian and Aloysius Ka-Lau Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, September 1986.

JU93. Mark B. Josephs and Jan Tijmen Udding. An overview of D-I algebra. In Mudge, Multinovic, and Hunter, editors, *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, volume I, pages 329–338. IEEE Computer Society Press, January 1993.

Kan95. Inhye Kang. *CTSM: A Formalism for Real-Time System Analysis Based on State-Space Exploration.* PhD thesis, Department of Computer and information Science, University of Pennsylvania, Philadelphia, PA 19104-6389, February 1995. Prospectus.

Koy92. R. Koymans. Specifying real-time properties with metric temporal logic. In Krishna M. Kavi, editor, *Real-Time Systems Abstractions, Languages, and Design Methodologies*, pages 88–132. IEEE Computer Society Press, 1992.

Kri92. Padmanabhan Krishnan. A calculus of timed communicating systems. *International Journal of Foundations of Computer Science*, 3(3):303–322, September 1992.

LA90. Shem-Tov Levi and Ashok K. Agrawala. *Real-Time System Design.* McGraw-Hill Book Company, New York, 1990.

Lad86. Peter Ladkin. Time representation: A taxonomy of interval relations. In *Proceedings of AAAI-86*, pages 360–366, 1986.

LBGG94. Insup Lee, Patrice Brémond-Grégoire, and Richard Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. In *Proceedings of the IEEE, Special Issue on Real-Time Systems*, January 1994.

LLPY97. Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems:compact data structure and state-space reduction. In *Proceedings of 80th IEEE Real-Time Systems Symposium*, pages 14–24, December 1997.

LM91. Michael R. Lowry and Robert D. McCartney, editors. *Automating Software Design.* AAAI Press and the MIT Press, 1991.

LY93. Kim G. Larsen and Wang Yi. Time-abstracted bisimulation: Implicit specifications and decidability. In *Proceedings of Mathematical Foundations of Programming Semantics (MFPS)'93*, pages 160–176, 1993.

MC90. Derek P. Mannering and Bernard Cohen. The rigorous specification and verification of the safety aspects of a real-time system. In *Proceedings of the Fifth Annual Conference on Computer Assurance (COMPASS 90)*, pages 68–85, New York, June 1990. IEEE.

Mil80. R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

Mil89. R. Milner. *Communication and Concurrency.* Prentice-Hall International, London, 1989.

MM91. Wenbo Mao and George J. Milne. An automated proof technique for finite-state machine equivalence. In *Proceedings of Cav '91*, pages 233–243, 1991.

Mol91. Faron Moller. Process algebra as a tool for real time analysis. In G. Birtwistle, editor, *Proceedings of the IV Higher Order Workshop*, pages 32–53, Berlin, Germany, September 1991. Springer-Verlag.

MP95. Oded Maler and Amir Pnueli. Timing analysis of asynchronous circuits using timed automata. In *Proceedings of Correct Hardware Design and Verification Methods. IFIP WG 10.5 Advanced Research Working Conference, CHARME 95*, pages 189–205, Berlin, October 1995. Springer Verlag.

MRM94. Chris J. Myers, Thomas G. Rokicki, and Teresa H.-Y. Meng. Automatic synthesis and verification of gate-level timed circuits. Technical Report CSL-TR-94-652, Stanford University, Computer Systems Laboratory, Department of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305-4055, December 1994.

MT92. Faron Moller and Chris Tofts. A temporal calculus of communicating systems. In Krishna M. Kavi, editor, *Real-Time Systems Abstractions, Languages, and Design Methodologies*, pages 242–256. IEEE Computer Society Press, 1992.

Pnu98. Amir Pnueli. Invited talk, 21st century engineering consortium workshop. Weizmann Institute of Science, DARPA/ITO Sponsored, March 1998.

RM94. T. G. Rokicki and C. J. Myers. Automatic verification of timed circuits. In *International Conference on Computer-Aided Verification*, pages 468–480. Springer-Verlag, 1994.

RR92. G.M. Reed and A. W. Roscoe. Timed csp: Theory and practice. In Krishna M. Kavi, editor, *Real-Time Systems Abstractions, Languages, and Design Methodologies*, pages 206–241. IEEE Computer Society Press, 1992.

Rus95. John Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1995. Also available as NASA Contractor Report 4673, August 1995, and to be issued as part of the *FAA Digital Systems Validation Handbook* (the guide for aircraft certification).

SJ94. Y. V. Srinivas and Richard Jüllig. Specware: Formal support for composing software. Technical report, Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304, Draft: Dec 11, 1994.

SS94. Oleg V. Sokolsky and Scott A. Smolka. Incremental model checking in the modal mu-calculus. In *Proceedings of CAV'94, LNCS 818*, June 1994.

SS95. Oleg V. Sokolsky and Scott A. Smolka. Local model checking for real-time systems. In *Proceedings of CAV'95*, 1995.

Ste94. Kenneth S. Stevens. *Practical Verification and Synthesis of Low Latency Asynchronous Systems.* PhD thesis, The University of Calgary, Calgary, Alberta Canada, September 1994.

SY96. Joseph Sifakis and Sergio Yovine. Compositional specification of timed systems. In *Proceedings of STACS 96. 13th Annual Symposium on Theoretical Aspects of Computer Science*, LNCS-1046, pages 347–359. Springer-Verlag, 1996.

TAKB96. Serdar Tasiran, Rajeev Alur, Robert P. Kurshan, and Robert K. Brayton. Verifying abstractions of timed systems. In *Proceedings of 7th International Conference on Concurrency Theory*, pages 546–562. Springer-Verlag, 1996.

TB97. Serdar Tasiran and Robert K. Brayton. Stari: A case study in compositional and hierarchical timing verification. In *Proceedings of the Computer Aided Verification Conference*, 1997.

Tsa87. Edward Tsang. Time structures for ai. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 456–461, Los Altos, CA, 1987. Morgan Kaufmann.

van90. R. J. van Glabbeek. The linear time - branching time spectrum. In J. C. M. Baeten and J. W. Klop, editors, *Concur 90*, pages 278–297, Berlin, 1990. Springer-Verlag.

Wan90. Yi Wang. Real-time behavior of asynchronous agents. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of CONCUR '90*, 1990.

Vita

Major Frank Charles Duane Young was born on 22 April 1955 in Choteau, Montana. He graduated from Choteau High School in 1973 and attended Carroll College, Helena, Montana from 1973 to 1974. He enlisted in the United States Air Force in October 1974. From January 1975 to October 1979 Major Young worked as an enlisted computer operator at Air Force Global Weather Central, Offutt AFB, Nebraska. From October 1979 to August 1984, Major Young was a computer operator and computer systems manager in the base data processing facility at Malmstrom AFB, Montana. From September 1984 to June 1987, Major Young attended Montana State University via the Air Force's Airman's Education and Commissioning Program. He graduated Magna Cum Laude with a Bachelors Degree in Computer Science. From July 1987 to October 1987, Major Young attended the Air Force Officer Training School; he was commissioned on 1 October 1987. After a short Technical School assignment at Keesler AFB Mississippi, Major Young was the Deputy Chief of Computer Operations in Cheyenne Mountain Complex, Colorado Springs, Colorado from March 1988 to August 1990. From September 1990 to May 1991, Major Young was a Software Engineer and Programmer Analyst on a software development project for a new Cheyenne Mountain system. From May 1991 to March of 1996, Major Young was a student at the Air Force Institute of Technology (AFIT). In December of 1992, he received his AFIT Master of Science degree in Computer Engineering. From March of 1996 to April 1999, Major Young was a Hardware/Software Systems Research Engineer in the Air Force Research Laboratory Avionics and Information Directorates. Currently, Major Young is the Operations Flight Commander at the Theater Air Command and Control Simulation Facility, Kirtland AFB New Mexico.

Permanent address: 110 1st Ave. S.W.
Choteau, Mt, 59422-0460

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
15-06-1999

2. REPORT TYPE
Ph.D. Dissertation

3. DATES COVERED *(From - To)*
Jan 1993-Jun 1999

4. TITLE AND SUBTITLE
TIMED SAFETY AUTOMATA AND LOGIC CONFORMANCE

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Frank C. D. Young

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2950 P Street
WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/DS/ENG/99-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
James S. Williamson
AFRL/IFTA
2241 Avionics Circle
WPAFB, OH, 45433-7334

(937)255-6653x3607

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
N/A

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Timed Logic Conformance (TLC) is used to verify the behavioral and timing properties of detailed digital circuits against abstract circuit specifications when both are modeled as Timed Safety Automata (TSA) with real-valued clocks. TLC is a bisimulation-style partial order relationship defined over TSA state space. TLC defines when one system is an acceptable implementation of another by asymmetric action-matching requirements for specification inputs and implementation outputs. TLC intuitively and pragmatically supports writing abstract specifications and verifying them against implementations. TLC scales up by substituting verified specifications for implementations and hierarchically verifying larger systems. The TLC verification process is more efficient than the circularly dependent assumes-guarantees verification methodology. The TLC verification methodology explicitly captures environmental timing properties in the system specification and automatically ensures they are satisfied in the TLC relation. The region-automata-based Timed Logic Conformance System (TLCS) implements TSA parallel composition and a TLC decision procedure. TLCS is used to hierarchically verify the STARI (Self-Timed at Receiver's Input) asynchronous circuit for communicating safely between clock-skewed systems.

15. SUBJECT TERMS
Formal Verification of Digital Electronics, Timed Safety Automata, Region Automata, Bisimulation, Partial Order Refinement, Calculus of Communicating Systems, Timing Verification.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Thomas C. Hartrum
Unclassified	Unclassified	Unclassified	UU	169	19b. TELEPHONE NUMBER *(Include area code)* (937)-255-3636x4581